JOT
JOURNAL OF
OBJECT TECHNOLOGY

Journal of Object Technology | **RESEARCH ARTICLE**

# A Query-based Approach for Verifying UML Class Diagrams with OCL Invariants

**Hao Wu***
*Computer Science Department, Maynooth University, Ireland

**ABSTRACT** Verifying whether a UML class diagram is consistent involves finding valid instances that provably meet its constraints defined in Object Constraint Language (OCL). Recent studies have shown that many existing tools and techniques not only can find valid instances but also pinpoint the conflicts among the OCL constraints. However, they do not scale well and are often unable to locate the conflicts when the number of OCL constraints significantly increases. In this paper, we present a novel approach that is capable of verifying UML class diagrams with a large number of OCL constraints. Our approach has two distinct features: (1) it provides a query language that allows users to choose parts of a UML class diagram to be verified. (2) a new algorithm that can handle an extreme size of OCL invariants via concurrent verification. We have implemented a new automated tool called: QMaxUSE. The evaluation results suggest that QMaxUSE has the potential to be adapted by industry and offers up to 30x efficiency improvement in verifying UML class diagrams with a large number of OCL constraints.

**KEYWORDS** Query, OCL, Concurrency

## 1. Introduction

In model-driven engineering (MDE), UML class diagrams are widely used to model structures of a system (Atkinson & Kühne 2003; Booch et al. 2005). For example, an entity of a system is typically depicted as a class, and relationships between different entities are represented as inheritance or associations. However, using a UML class diagram alone is not able to express textural constraints. To complement this, Object Constraint Language (OCL) is introduced and used by modeling practitioners to express additional constraints that cannot be captured by the UML graphical notation. This leads to a system that is modelled with a UML class diagram with OCL constraints more formal. However, this also means that reasoning or verifying the consistency of a UML class diagram becomes very challenging (Berardi et al. 2005; Queralt & Teniente 2006).

Formally, verifying the consistency means that checking whether a valid instance can or cannot be generated from a UML class diagram and its OCL constraints. If it cannot be generated (found), then a UML class diagram is said to be inconsistent. This implies that there exists at least one conflict in the structural and OCL constraints defined in the UML class diagram. To tackle this challenge, many approaches and techniques have been proposed (Gogolla et al. 2018; Queralt et al. 2012a; Maraee & Balaban 2007; Balaban & Maraee 2013; Wu & Farrell 2021). There are still two main challenges remaining: (1) When a UML class diagram is inconsistent, many existing tools are unable to pinpoint the set of OCL constraints that cannot be satisfied. (2) When the number of OCL constraints significantly increases, the existing tools and techniques do not scale well(Wu & Farrell 2021; Wu 2017b).

Our previous tool MaxUSE is designed to verify a UML class diagram by providing the user with two pieces of useful information (Wu & Farrell 2021; Wu 2017a,b): (1) the set of achievable features in a UML class diagram 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. However, when the number of OCL invariants increases MaxUSE struggles to handle complex formulas and often times out just like many other tools.

**JOT reference format:**
Hao Wu. *A Query-based Approach for Verifying UML Class Diagrams with OCL Invariants*. Journal of Object Technology. Vol. 21, No. 3, 2022.
 http://dx.doi.org/10.5381/jot.2022.21.3.a7

The performance and scalability of a tool is particularly important for industries (Ali et al. 2014; Iqbal et al. 2012; Garry & Balfe 2012). This is because they typically have models with a large number of OCL constraints. Hence, our aim in this paper is to tackle the following two challenges:

1. Provide an interactive verification that allows users to incrementally verify the consistencies of their models by selecting different parts of their design for analysis.
2. Offer a scalable approach that can pinpoint the conflicting constraints when a model has a large number of complex OCL invariants.

In this paper, we propose a query based approach to tackle these two challenges. Our approach provides two distinct features: (1) a query language that easily allows users to select parts of a UML class diagram to be verified. (2) a novel algorithm that can concurrently verify OCL invariants.

Verifying a class diagram based on queries has three main advantages: (a) It is interactive and this is particularly useful when a user would like to ensure the partial consistency of their design before extending it to a more complete design. (b) When the number of OCL constraints significantly increases, these constraints can be decomposed into several queries that can be verified concurrently. (c) Each query issued by users can be stored as a procedure is available to be used as a test suite later. This helps users to narrow down the part of their design that is inconsistent.

To demonstrate the effectiveness of our approach, we implement a new tool named QMaxUSE and we evaluate its effectiveness in answering three research questions. First, we compare QMaxUSE against a number of existing OCL tools. We highlight how we ascertain QMaxUses's capabilities for finding conflicts for large numbers of OCL invariants. Second, we deployed QMaxUSE to our industry partner, They used it with their sample models and from the results provided feedback that we have analysed. The responses were positive and this affirms that QMaxUSE has a great potential to be adapted by the software industry. Last, we evaluate QMaxUSE on a comprehensive benchmark to show its true performance. The results suggest that QMaxUSE significantly outperforms MaxUSE and illustrates the effectiveness of our query-based approach.

We believe this paper significantly advances verification techniques in the area and its contributions can be summarised as follows:

1. We propose a new query language that allows users to pick the parts of a UML class diagram including OCL invariants to be verified. In this way, users are able to incrementally verify different parts of their class diagrams by issuing different queries. Each query can be considered as a single test case for testing parts of their design (Section 3).
2. We propose a new algorithm that is capable of handling large numbers of complex OCL invariants. This algorithm works by decomposing a UML class diagram that is annotated with OCL invariants into different queries that can be verified concurrently through an SMT solver (Section 4).
3. We implement our approach into a new tool called: QMaxUSE. This tool is fully automatic, open-source and capable of verifying a large number of OCL invariants (Section 5).
4. We evaluate QMaxUSE by answering three research questions (Section 6). Our evaluation results suggest that QMaxUSE has a great potential to be adapted by the software industry and it significantly outperforms our previous tool MaxUSE (Wu 2017b; Wu & Farrell 2021).

In order to frame our subsequent discussions, we begin by introducing a motivating example in Section 2.

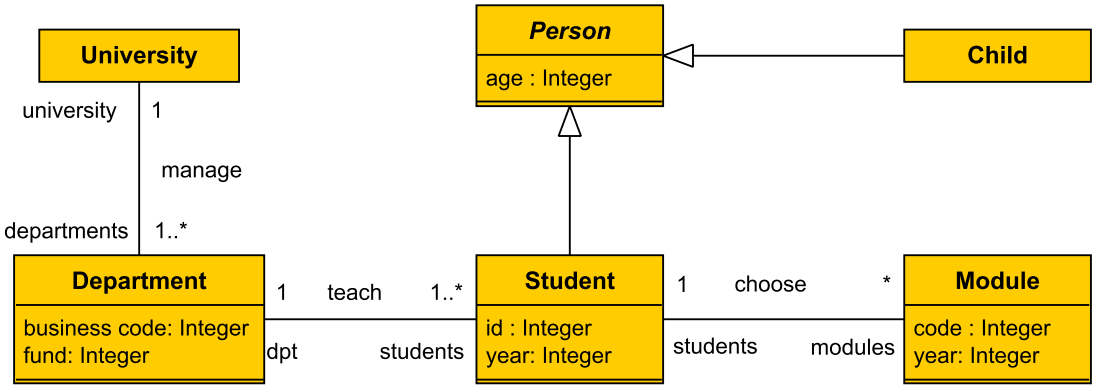
## 2. A Motivating Example

In this section, we provide an example that is used throughout this paper to illustrate how our approach works at a high level. This example was first introduced in (Wu 2017a,b) and later extended in (Wu & Farrell 2021). It models a university student that can select multiple modules to study. This is shown in Figure 1. Furthermore, this UML class diagram has 8 OCL class invariants. These invariants impose additional constraints. For example, $inv5$ indicates a student can select modules that are available only in their year [1]. A department must have some research students and non-research students. This constraint is reflected in $inv6$.

Unfortunately, this model is not consistent and a verifier cannot find a valid instance for this model. This is because there are two conflicts among the 8 OCL class invariants. The first conflict is easy to spot, and it occurs between $inv1$ and $inv2$. $inv1$ indicates that every person's age is between 0 and 18, while $inv2$ requires that every student in a university is over 18. The second conflict is not straightforward and caused by the invariants: $inv5$, $inv6$, $inv7$ and $inv8$. This is because a department allows some research and non-research students ($inv6$) to choose some modules ($inv7$) in their corresponding year ($inv5$). However, $inv8$ indicates that modules are only available for non-research students ($inv8$: between year 1 and 5).

**Observation.** Based on the two conflicts, we could make two observations. We first observe that the conflict that occurs between $inv1$ and $inv2$ places two opposite constraints on the same attribute *age* in the class *Person*. This means that we could put $inv1$ and $inv2$ into one group. Similarly, the second conflict occurs in the four invariants that cover two attributes (*year* in the *Student* and *Module* class) and one navigation ($s.modules$) for two different classes. Thus, we could put these invariants together into another group. Secondly, we observe that the invariants in both groups are independent from each other. In other words, we could consider two groups as two queries that can be checked separately by a verifier. Based on these observations, the 8 OCL class invariants along with the UML class diagram (in Figure 1) can hence be decomposed into three queries (groups) that can be verified concurrently. These groups can be visualised as a dependency graph that is shown in Figure 2.

[1] In this example, numbers 1 to 6 are used to distinguish a student's year. Students that are in year 6 are considered as research students.




```
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
```

**Figure 1** A UML class diagram with the 8 OCL class invariants shows how the students in each department can choose multiple modules to study.

## 3. A Query Language

In this section, we introduce a query language that can be used to choose parts of a UML class diagram to be verified. The syntax of our query language is shown in Figure 3.

### 3.1. Class, Attribute and Association Selection

A *query* expression must use a *select* statement that allows users to choose multiple features from a UML class diagram. A *feature* here may include a *class*, an *attribute*, an *association* or an *invariant*. For example, the following query selects the *University*, *Department* and *Student* class along with an association *teach* from the UML class diagram in Figure 1.

```
select University, Department, Student
, Department:teach:Student
```

A wild character * can be used within a select statement to choose many features. When a feature is selected, its owner is also implicitly selected. For example, the following query

```
select Student.*
```

selects the *Student* class and all of its attributes (Figure 1). However, the owner of the *age* and *gender* attributes are the *Person* class. Hence, the *Person* class is also selected. Users can also issue a query that explicitly selects all child or parent classes of a specified class. This can be done using our modifiers *upward* and *downward*. Modifier *upward* enables our selection algorithms to select all parent classes of a class while in contrast *downward* facilitates selection of all child classes of a class. For example, the following query selects three classes (and their attributes): *Person*, *Student* and *Child*.

```
select downward Person.*
```

Currently, modifiers *upward* and *downward* are experimental. A selection statement that uses *upward* or *downward* only selects classes and attributes. The number of instances of an abstract class that has no selected (non-abstract) child classes is by default set to 0. This allows us to reason the number of instances of abstract and non-abstract classes.

### 3.2. Invariant Selection

An OCL invariant is typically defined under a specified class to indicate which class it belongs to. Based on this principle, our query language requires a specified class name in front of each selected OCL invariant. This is

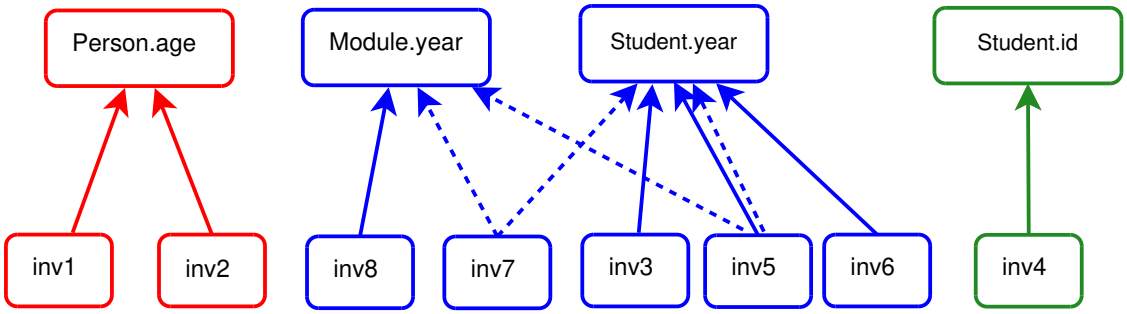



**Figure 2** A dependency graph showing the three groups of invariants computed by QMaxUSE. Each group is marked with a different colour and can be verified separately. The solid line arrows indicate the attributes of different classes are constrained by different OCL invariants. The dashed arrows represent the classes or attributes that are constrained through a navigation expression within an OCL invariant (*inv*5, *inv*7).

achieved by using the *with* expression. For example, the query `select Module with Student::*` selects the *Module* and *Student* class along with all the OCL invariants (*inv*2 to *inv*7) defined under the *Student* class (Figure 1). Since the invariants are imposed on different attributes, these attributes (used within these invariant expressions) are also implicitly selected (*Student.age*, *Student.year*, *Student.id*, *Module.year*). Furthermore, both *inv*5 and *inv*7 use a navigation expression (*s.modules*) through the *choose* association, so *choose* is also selected here.

In some scenarios, a user may select as many features or invariants as they like except for a few ones. To accommodate these scenarios, we add a syntactic sugar into our grammar to avoid a situation that a user must explicitly specify features or invariants one by one. This is achieved by using the *but* expression. The *but* expression allows users to exclude multiple features or invariants from a query. For example, the following two queries essentially select the same set of features and invariants from Figure 1. In other words, they both exclude *inv*4 and *inv*7 from selection.

```
select Person with Student::* but Student::inv4,
Student::inv7
select Person with Student::inv2, Student::inv3,
Student:inv5, Student::inv6
```

### 3.3. Query Pattern

By using a wild character *, a user is able to issue a query using one or more query patterns to select a collection of features. The rule here is that our selection algorithms only select those features (classes, attributes and associations) that are explicitly specified in a select statement and implicitly selects those that are used by those features. Table 1 summarises a list of these query patterns and their semantics.

### 3.4. Joint Query

A user may issue multiple queries first and then join them later to verify a *joint query* that covers other parts of a class diagram. A joint query in our language is defined by one of the three joint operators: intersection (&), union(+) and difference(−). For example, the query `select Department.*+ select Module.*` is a joint query that uses a union operator to select all the attributes defined in the *Department* class and *Module* class.

To facilitate joining multiple queries into a single joint query, we allow users to name their individual query using an *alias* expression. For example, the following two queries are associated with alias *q*1 and *q*2, respectively.

```
select Person.*, Student.*, Module.* with Student::*,
Module::* as q1
select University.*, Department.*, Module.* as q2
```

A joint query (intersection) can then be easily formed by using `q1 & q2`. In this case, the *Module* class and all its attributes are selected.

### 3.5. Query Module

We provide a query module that allows users to store their predefined queries so that they can be used as test cases to cover parts of their UML class diagram. A query module is saved in a specification file and it must contain at least one query. For example, the following query module (*QuerySet*) consists of three queries. Each query covers a part of a UML class diagram in Figure 1.

```
module QuerySet
select Person.*, Student.* with Person::*, Student::* as t1
select Student.*, Module.*, Student:*:* with Student::*,
Module::* as t2
select Student.id, Department:*:* with Student::inv4 as t3
end
```

The first query (*t*1) acts as a test case that covers the features defined in the *Person* and *Student* class including their OCL invariants (*inv*1 and *inv*2). By verifying these features (covered by *t*1), one can uncover the conflict between the two invariants *inv*1 and *inv*2. The second query (*t*2) covers the set of features about how a student chooses a module including the OCL invariants (*inv*2-*inv*8) defined under the *Student* and *Module* class. The verification of this set of features can reveal the

$$
\begin{array}{lll}
\langle Expr\rangle & ::= & \langle QueryExpr\rangle \\
 & & \mid \langle JointQuery\rangle \\
 & & \mid (\langle QueryModule\rangle)^{*} \\
 & & \\
\langle JointQuery\rangle & ::= & \langle QueryExpr\rangle \textbf{ + } \langle QueryExpr\rangle \\
 & & \mid \langle QueryExpr\rangle \textbf{ - } \langle QueryExpr\rangle \\
 & & \mid \langle QueryExpr\rangle \textbf{ \& } \langle QueryExpr\rangle \\
\langle QueryExpr\rangle & ::= & \textbf{select } \langle FeatureExpr\rangle(\textbf{, } \langle FeatureExpr\rangle)^{*} \\
 & & \\
 & & (\langle withExpr\rangle)? \ (\langle butExpr\rangle)? \ (\textbf{as IDENT})? \\
\langle FeatureExpr\rangle & ::= & \langle ClassExpr\rangle \mid \langle AttrExpr\rangle \mid \langle AssocExpr\rangle \\
\langle ClassExpr\rangle & ::= & \langle NameSpace\rangle \\
 & & \\
\langle AttrExpr\rangle & ::= & \langle NameSpace\rangle \textbf{ . } \langle NameSpace\rangle \\
\langle AssocExpr\rangle & ::= & \langle NameSpace\rangle \textbf{ : } \langle NameSpace\rangle\textbf{:}\langle NameSpace\rangle \\
 & & \\
\langle withExpr\rangle & ::= & \textbf{with } \langle InvExpr\rangle(\textbf{, } \langle InvExpr\rangle)^{*} \\
\langle InvExpr\rangle & ::= & \langle NameSpace\rangle\textbf{::} \langle NameSpace\rangle \\
\langle butExpr\rangle & ::= & \textbf{but } (\langle FeatureExpr \mid InvExpr\rangle)(\textbf{, } \langle FeatureExpr \mid InvExpr\rangle)^{*} \\
 & & \\
\langle NameSpace\rangle & ::= & \text{IDENT} \mid \textbf{ *} \\
\langle QueryModule\rangle & ::= & \textbf{module IDENT} \\
 & & (\langle QueryExpr\rangle \mid \langle JointQuery\rangle)^{+} \\
 & & \textbf{end}
\end{array}
$$

**Figure 3** The syntax of our query language.

second conflict. That is, a department allows some research and non-research students ($inv6$) to choose some modules ($inv7$) in their corresponding year ($inv5$). However, $inv8$ indicates that modules are only available for non-research students ($inv8$: between year 1 and 5). The last query ($t3$) takes care of the rest of the features and the verification of this query does not find any conflicts. Hence, the three queries can be considered as three test cases covering different parts of a UML class diagram.

### 3.6. Query Verification

In this section, we describe a verification procedure that can automatically verify a query issued by a user via SMT solving. Our query engine executes a query issued by a user and yields a query result. In fact, the result returned from a query execution is a set containing relevant classes, attributes, associations and OCL invariants. We consider this set as a *query result* that reflects parts of a UML class diagram collected by a query. We define a query result ($q_r$) as a 4-tuple:$q_r = \langle C, A, O, I\rangle$ where $C$ is a set of classes, $A$ is a set of attributes, $O$ is a set of associations and $I$ is a set of OCL invariants.

A high-level structure of our algorithm (QueryVerification) for verifying a query is shown in Algorithm 1. To correctly collect the relevant features from a query, our selection algorithm executes a query ($q$) by traversing the abstract syntax tree (AST) of a query and iteratively adds every specified feature into a query result ($q_r$). For an OCL invariant, the algorithm [2] also traverses the AST of an OCL invariant and implicitly collects the classes, attributes and associations used within that OCL invariant. Once a $q_r$ is generated, we can cast it to a set of first-order formulas that can be verified by an SMT solver. The encodings of a feature is similar to the ones described in (Wu & Farrell 2021). For example, we use uninterpreted functions

[2] Currently, our algorithm skips operational contracts during traversal.

| Query Pattern | Semantics |
|---|---|
| $A.*$ | all attributes of class $A$ (including attributes that are inherited from parent classes). All parent classes of $A$ including class $A$ are also implicitly selected. |
| $*.*$ | all classes and attributes in the current UML class diagram. |
| $A:*:B$ | all associations between Class $A$ and $B$, Class $A$ and $B$ are also implicitly selected. |
| $A:*:*$ | all associations that have Class $A$ at one end. Class $A$ is also implicitly selected. |
| $*:*:*$ | all associations in the current UML class diagram. Classes from association-ends are also implicitly selected. |
| $A::*$ | all invariants under Class $A$ are selected. Features (classes, attributes and associations) used in invariants are implicitly selected. |
| $*::*$ | all invariants in the current UML class diagram. |

**Table 1** A summary of query patterns and their semantics

to encode classes, attributes and linear integer inequalities to capture the multiplicities at an association-end. For an OCL invariant, we traverse its AST and generates a formula by using a combination of first-order theories.

**Algorithm 1 QueryVerification**

**Input** : A query that contains a set of features($q$).
**Output** : A set that contains conflicting features ($S$).
$S \leftarrow \emptyset$;
$q_r \leftarrow q.execute()$; //generate a query result $q_r$
/* generate a set of first-order formulas $\Phi$ for each feature in a query result*/
$\Phi \leftarrow FOLTranslate(q_r.cls(), q_r.attr(), q_r.assoc(), q_r.inv())$;
**foreach** $\phi_i \in \Phi$ **do**
  $\phi_i \leftarrow label(\phi_i)$; //label each formula.
**end**
**if** $(SMTSolve(\Phi) == \textbf{UNSAT})$ **then**
  **foreach** $\phi \in Solver.cores()$ **do**
    $S.add(Interpret(\phi.label()))$;
  **end**
**else**
  //report selected features are consistent
**end**
**return** $S$;

After the translation to first-order formulas, we label each individual formula ($\phi$) so it can be tracked and interpreted (back to a feature or an OCL invariant) when the SMT solving finishes. If $\Phi$ is unsatisfiable, the algorithm retrieves the unsat cores from an SMT solver and pinpoints the conflicts among the OCL invariants by interpreting each formula's label. Otherwise we report that the set of features selected ($q_r$) is consistent.

## 4. Concurrent Verification

Since we can verify each query using the procedure (QueryVerification) shown in Algorithm 1, we may issue a series of queries that cover different parts of a UML class diagram and verify them separately. This is particularly *powerful* when the number of OCL invariants significantly increases. Our concurrent verification algorithm takes a UML class diagram with its OCL invariants as input and outputs a set $S$ that contains all possible conflicting OCL invariants. This algorithm relies on a decomposition procedure that is capable of decomposing a set of OCL invariants into different queries. This algorithm (ConcurrentVerification) is shown in Algorithm 2.

The core of this concurrent verification algorithm is a procedure *Decompose*. This procedure takes in a model [3] as its input and outputs a group of queries that can be verified concurrently. The *Decompose* procedure consists of three parts.

**Algorithm 2 ConcurrentVerification**

**Input** : A UML class diagram annotated with OCL invariants ($model$)
**Output** : A set of conflicting features causing inconsistencies ($S$).
$Q \leftarrow Decompose(model)$; //decompose a model into a group of queries.
$S \leftarrow \emptyset$;
**foreach** $q_i \in Q$ **do**
  $ThreadManager.start(S.add(QueryVerification(q_i)))$
**end**
**return** $S$;

The first part of our *Decompose* procedure (shown in Algorithm 3) constructs a (dependency) graph that indicates an attribute that is imposed by one or more OCL invariant(s). To build a dependency graph, our algorithm uses a map structure ($map_g$) to remember the incoming edges. Algorithm 3 loops through every attribute defined in a class and checks if it is used by an OCL invariant. If an attribute ($attr$) is used in an OCL invariant ($inv$), the algorithm then continues to check if it already exists in our $map_g$. If it exists, then we add a new

[3] Here, we consider a model as a UML class diagram annotated with OCL invariants

| | |
|---|---|
| Module.year | {inv5, inv8} |
| Student.year | {inv3, inv6, inv5} |
| Student.id | {inv4} |
| Person.age | {inv2, inv1} |

**Table 2** A map structure ($map_g$) containing a list of entries after computing Algorithm 3 on the OCL invariants defined in Figure 1.

incoming edge. Otherwise, we create a new node in our graph with the name of this attribute ($attr$) and add a new edge.

**Algorithm 3 Decompose(Part 1)**

```
foreach cls ∈ model.classes() do
    foreach attr ∈ cls.attributes() do
        foreach inv ∈ model.invariants() do
            if IsUsed(attr, inv) then
                if map_g.contains(attr) then
                    map_g.get(attr).add(inv); //add an edge to
                    the graph
                else
                    s ← ∅; //create a new set s.
                    s.add(inv);
                    map_g.add(attr, s); // create a new node in
                    the graph
                end
            end
        end
    end
end
```

Hence, after computing the 8 OCL invariants in Figure 1 using Algorithm 3, we get the following entries in our map structure. For example, the first entry means that the *year* attribute defined in the *Module* class is used in both invariants $inv5$ and $inv8$.

The second part (shown in Algorithm 4) of our *Decompose* procedure first creates a new set $G$ that is used to store a group of invariants. It then checks every entry in $map$, merges two entries that have common invariant(s) into one and adds each group of invariants (created in $map$) into $G$. The intuition behind this is that if two entries have at least one common invariant, this implies that two different attributes are used within an invariant. Hence, other invariants that use these two attributes must also be included because they may have a chance of causing a conflict. For example, the first and second entry in Table 2 have a common invariant $inv5$. This means that the two attributes Student.year and Module.year are used in $inv5$. Thus, the first two entries in Table2 are merged. Hence, our new set $G$ now contains three sets of invariants shown in Table 3.

| | |
|---|---|
| Group 1 | {inv3, inv6, inv5, inv8} |
| Group 2 | {inv4} |
| Group 3 | {inv2, inv1} |

**Table 3** A set $G$ containing three groups of invariants.

**Algorithm 4 Decompose(Part 2)**

```
G ← ∅; //create a new set G.
foreach e_i ∈ map //entry e_i do
    foreach e_j ∈ map //entry e_j do
        /*merge two entries that share common invariant(s).*/
        if (e_i ≠ e_j) and (e_i ∩ e_j ≠ ∅) then
            e_i = e_i ∪ e_j
            map.remove(e_j)
        end
    end
    G.add(e_i) //G contains entries from map including merged
    ones.
end
```

The final part of our *Decompose* procedure is shown in Algorithm 5. This part takes care of the OCL invariants that use a navigation expression. A navigation expression represents a collection of objects whose type is a class. We use another map structure $map_t$ to remember a collection of object types used in a navigation expression. For example, after constructing $map_t$ for the UML class diagram in Figure 1, it contains an entry of $[Module, \{inv5, inv7\}]$[4]. This is because both invariants ($inv5$ and $inv7$) use a navigation expression ($s.modules$). We then go through each group of invariants ($g$) to check if any group contains at least one invariant recorded by $map_t$. If it does, this implies that these invariants (recorded by $N$) impose constraints on a common set of features covered by $g$. Hence, we merge them, update $G$ and mark the corresponding entry in $map_t$. For example, $\{inv5, inv7\}$ is merged with $Group\ 1$ in Table 3 resulting in Table 4.

Next, we gather each entry in $map_t$ that is not marked into a new group. This is because we know that these invariants (returned from $map_t.get(cls)$) do not share common features imposed by existing groups of invariants. Hence, they can be safely gathered into a separate group. Then, we add the invariants that are not covered by our existing groups of invariants into $G$. Finally, $G$ contains $k$ groups of invariants. To produce a query for each group, we generate a *select* statement for every invariant including those features that are used within an OCL expression. The number ($k$) of queries in $Q$ represents that a *model* can be verified with $k$ threads. For example, the UML class diagram in Figure 1 can be decomposed into 3 queries for the 3 entries in Table 4.

[4] $map_t.get(Module)$ returns $inv5$ and $inv7$.

| | |
|---|---|
| Group 1 | {inv3, inv6, inv5, inv8, inv7} |
| Group 2 | {inv4} |
| Group 3 | {inv2, inv1} |

**Table 4** A set $G$ that has three groups of invariants. This implies that the UML class diagram in Figure 1 can be decomposed into 3 queries and verified concurrently using 3 threads.

**Algorithm 5 Decompose(Part 3)**

**foreach** $cls \in map_t.entries()$ **do**
  **foreach** $g \in G$ **do**
    **if** $map_t.get(cls) \cap g \neq \emptyset$ **then**
      $g \leftarrow g \cup map_t.get(cls)$; //returns a set of invariants
      $G.update(g)$;
      //mark this entry in $map_t$
      *break*;
    **end**
  **end**
**end**
**foreach** $e \in map_t.entries()$ **do**
  **if** $e$ *is not marked* **then**
    /*add those invariants do not share features with other ones.*/
    $G.add(map_t.get(e))$;
  **end**
**end**
//Create a new group $g'$ for the remaining invariants
$G.add(g')$; //add remaining invariants
$Q \leftarrow \emptyset$; //create an empty set of queries
**foreach** $g \in G$ **do**
  /*generate a query for each group*/
  $Q.add(GenerateQuery(g))$;
**end**
**return** $Q$;

Though our *Decompose* procedure takes navigations into account, it may not cover "all possible" scenarios that could escape detection by our *Decompose* procedure. Currently, we are investigating more scenarios and plan to improve the *Decompose* procedure in the future. However, the *Decompose* procedure presented in this Section at the very least covers many of the essential usages of navigations in an OCL invariant.

## 5. Implementation

We have built a prototype tool called QMaxUSE(Wu 2022). QMaxUSE is built on top of MaxUSE (Wu 2017b; Wu & Farrell 2021). QMaxUSE implements our query language in Section 3 and a concurrent verification algorithm in Section 4. The overall architecture of QMaxUSE is shown in Figure 4. QMaxUSE is fully automatic, written in Java and consists of nearly 33$k$ lines of code. Approximately 3.5$k$ lines of code are dedicated to the new verification algorithms. QMaxUSE can be run on Windows 10, Linux (Ubuntu) and MacOS (Big Sur). QMaxUSE is a command-line based verification tool. It is open-source and currently available at:

https://github.com/classicwuhao/qmaxuse

QMaxUSE has four layers: front-end, query engine, translation and solver. The first three layers can be considered as QMaxUSE's query compiler that is mainly responsible for parsing OCL, executing queries and generating corresponding SMT formulas for the solver layer.

**Front-end.** At the front-end layer, QMaxUSE uses parsers from USE to generate ASTs (abstract syntax trees) for a class diagram and OCL invariants. QMaxUSE provides a simple query language that allows users to choose a part of a class diagram and its OCL invariants to be verified. To parse a query issued by a user, we have designed and implemented a query parser. This parser is able to read multiple queries simultaneously from a specification file and produce corresponding ASTs.

**Query Engine.** QMaxUSE's query engine uses a set of selection algorithms to traverse the ASTs generated from the front-end layer to produce a query result. A query result essentially contains a set of classes, attributes, associations and OCL invariants to be verified. At this layer, QMaxUSE also provides a *specialised algorithm* (Decomposer) that is able to decompose a class diagram along with OCL invariants into a set of different queries. These queries can then be verified concurrently using a query verification procedure.

**Translation.** At the translation layer, QMaxUSE uses a *first-order translator* to translate a query into a set of first-order formulas that can be verified by the SMT solver. The translation here is similar to the one described in (Wu & Farrell 2021). To translate an OCL invariant, we traverse its AST and use a dedicated engine to generate a well-formed SMT2 formula.

**Solver.** We design a new interface (*SolverManager*) to reduce the interaction overhead between the translation and solver layer. This interface is able to directly interact with an SMT solver and can be extended to multiple SMT solvers. Currently, QMaxUSE uses Z3 as its default SMT solver and we plan to support multiple SMT solvers in the future (De Moura & Bjørner 2008).

Figure 5 shows the verification result returned from QMaxUSE for a query:

```
select Person.*, Student.* with Person::inv1, Student::inv2
```

QMaxUSE's query engine selects the relevant features and launches our query verification algorithm (Algorithm 1) to formally verify this query. The result returned from QMaxUSE shows that there is a conflict between two invariants and the *Student* class in Figure 1.

Figure 6 shows a screenshot of running QMaxUSE. In fact, this screenshot shows the verification results from our concurrent verification algorithm (Algorithm 2 described in Section 4). QMaxUSE uses 3 threads to verify the UML class diagram in Figure 1 and pinpoints 2 conflicts. Note that QMaxUSE adds one additional axiom during the translation to SMT. This axiom

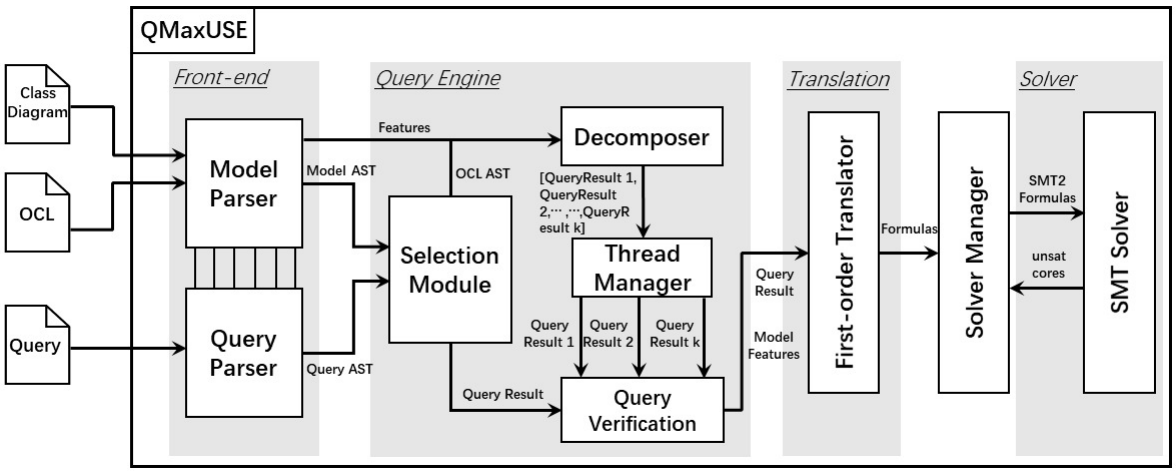



**Figure 4** The overall architecture of QMaxUSE.

```
QMaxUSE> $select Person.*, Student.* with Person::inv1, Student::inv2
Launching QueryCompiler...
Class: Person is added.
Attributes: [age : Integer, gender : Gender] are selected.
Class: Student is added.
Attributes: [id : Integer, year : Integer, age : Integer, gender : Gender] are selected.
Attributes: [age : Integer, gender : Gender] are selected.
========Selected Classes========
[Student, Person]
========Selected Attributes========
[Person.gender Student.year Student.id Person.age ]
========Selected Associations========
[]
========Selected Invariants========
[Student::inv2 Person::inv1 ]
=======Used Attributes========
age->{ Student::inv2 Person::inv1 }

z3 solver is picked.
verifying query (Linux) start...
Solving Finished from query.
unsat
cores: { inv2  inv1  Student  }
Time elapsed:36 ms

QMaxUSE>
```

**Figure 5** A screenshot showing the verification result returned from QMaxUSE for a single query.

```
QMaxUSE> qverify
Launching QueryCompiler...
z3 solver is picked.
verifying thread0 (Linux) start...
Solving Finished from thread0.
unsat
cores: { inv6  inv5  inv8  inv7  }
Time elapsed:42 ms

verifying thread1 (Linux) start...
Solving Finished from thread1.
unsat
cores: { inv2  inv1  Student  }
Time elapsed:23 ms

verifying thread2 (Linux) start...
Solving Finished from thread2.
sat
Time elapsed:25 ms

{ core 0: inv6  inv5  inv8  inv7  }
{ core 1: inv2  inv1  Student  }
Total conflicts: 2
Total Time Spent (3 threads): 132 ms.
QMaxUSE>
```

**Figure 6** A screenshot showing the verification results returned from QMaxUSE for the UML class diagram in Figure 1.

requires that every-non abstract class must be instantiated at least once. If a user wishes to explicitly specify the number of instances of a class (including an abstract class), we recommend to use *size* operation. This allows QMaxUSE to use a separate encoding that is solely focusing on the number of instances without intervening other axioms. Hence, the *Student* class is also included in *core* 1 in Figure 6. In other words, removal of either *inv*1 or *inv*2 makes the *Student* class instantiable.

## 6. Evaluation

We evalue QMaxUSE by answering the following 3 research questions:

- **RQ1. How does QMaxUSE compare to other existing tools?**
- **RQ2. Does QMaxUSE have the potential to be applied to industry-size problems?**
- **RQ3. What is the true performance of QMaxUSE?**

### 6.1. Answer to Research Question 1 (RQ1)

To answer RQ1., we reviewed the literature in this area and have found that there are a large number of approaches and techniques that have been proposed. However, most of them are evaluated without using a comprehensive benchmark (Cadoli et al. 2007; Cabot et al. 2014; Garis et al. 2011; Kuhlmann et al. 2011; Balaban & Maraee 2013). This makes it particularly difficult for us to assess their performance and scalability against QMaxUSE. To effectively compare it to existing tools, we decide to select those have been highly cited by others or where their tools are available to download. We studied their approaches and examine their tools by running them with the provided examples to assess their capabilities.

Table 5 summarises our selected features and comparison results. We choose these features mainly because we believe they represent a collection of essential features that an OCL verification or analysis tool should possess (Gogolla et al. 2013). For example, an OCL verification tool should at least be able to cover a range of OCL language features and return relevant

information if there exists some OCL constraint conflicts. We firmly believe these features are important to help users to better understand and improve their models (Wu & Farrell 2021).

Most of the tools we selected are capable of verifying the consistencies of a UML class diagram annotated with OCL invariants except for CD2Alloy. They support a range of OCL language features: quantifiers, collections, arithmetic & logic operators and navigations. Currently, QMaxUSE does not support verification of OCL operational contracts [5]. From Table 5, we can now identify two major limitations in these tools. (1) many tools can only process a small sample of OCL invariants and do not provide a benchmark for evaluating much larger and complex OCL invariants. (2) when a UML class diagram is inconsistent, they are unable to pinpoint conflicting OCL invariants. In comparison to these tools, our tool QMaxUSE has an advantage of providing a comprehensive benchmark with a large number of OCL invariants containing conflicting ones among them. The complexities and size of OCL invariants that QMaxUSE can handle are shown in the benchmark evaluation of Section 6.3 (our answer to RQ3).

From RQ1, we find that it is a really difficult task to assess the scalability and performance of existing OCL analysis tools without using a common benchmark (Wu 2018). Thus, $\geq 50$ in the Table 5 indicates the number of OCL invariants available for assessing a tool. It does not indicate a particular tool's scalability because their true performance cannot be decided based on the available OCL invariants provided. In general, this is mainly caused by two problems. (1) Most of the tools do not provide a click-to-run feature and require a number of steps for their configuration and the installation of many libraries. This significantly reduces the usability of a tool. (2) Many tools only provide a small number of sample models for testing and evaluation. If users would like to gain a sense of the performance capability of a tool, they either have to manually write a large number of OCL invariants themselves or find a model that is already built in a format that can be accepted by the tool. This can immediately reduce the level of interests in using or taking time to explore the capabilities of the tool. In contrast to the tools in the Table 5, QMaxUSE is easy to install and use. It does not require users to configure or install libraries and provides a full benchmark so that a user can immediately gain a sense of QMaxUSE's features and performance. More importantly, we welcome all kinds of feedbacks from users to guide us to improve QMaxUSE's usability and performance.

### 6.2. Answer to Research Question 2 (RQ2)

We demonstrated QMaxUSE to a team of engineers from Sun Yat-sen University Cancer Center (SYSUCC) [6]. This team has a total of 56 members and most of them are familiar with basic accounting rules. They are mainly responsible for testing and maintaining payment systems at SYSUCC. Part of their payment system was originally modelled using UML by a third-party vendor. Hence, they are familiar with basic UML and OCL notations. Currently, they are planning to upgrade their IT infrastructure including payment systems. We presented a short tutorial about QMaxUSE and distributed it to them. They tried and tested QMaxUSE on three sample models they had recently built. Two of them were originally from the initial design (that had been created many years ago) to model part of their payment systems. UML class diagrams were used for code generation but not for OCL invariants that are mainly used for design purpose. Most of the OCL invariants (about 115) defined for these two models deal with numerical constraints about payment information and transactions. One of the their models uses a significant number of string constraints (about 150) to model their invoice handling process. For their sample models, QMaxUSE exhibited its capabilities by handling more than 100 invariants and successfully finding two design flaws (conflicting constraints) in one of their models. The summary of the feedback received from these engineers is shown in Figure 7.

Before using QMaxUSE, the majority of this team from SYSUCC thought there is a huge gap between the current verification tools created in academia and the requirements of industry. After testing QMaxUSE on their sample models, 49 (88%) of them believe that QMaxUSE is capable of minimising this gap by handling larger models than the tools they had previously used. They think the query language provided by QMaxUSE is particularly useful. This is because they now can issue different queries and each of them can be considered as a test case for testing/verifying different parts of their designs. $4(7\%)$ of them think QMaxUSE is not suitable for some of the models that have string constraints such as a particular format for an invoice title. Currently, QMaxUSE does not provide string reasoning and skips an OCL invariant that uses a string type. Hence, we plan to integrate QMaxUSE with an efficient string solver. $3(5\%)$ of people are unsure about QMaxUSE because they are not convinced that whether UML and OCL could be useful for building complex models.

In terms of QMaxUSE's usability, a few engineers commented that it would be much better to explicitly display selected features before launching a verification procedure for interaction purposes. This is because they would prefer logging the selected features and precisely knowing which part of the UML has been chosen. Hence, we modified QMaxUSE's selection algorithms so that it now shows the features covered by a query.

In summary, QMaxUSE is demonstrated to live up to the expectation that it is capable of handling large number of OCL invariants and has a great potential to be adapted for industry-size problems. The team from SYSUCC is now considering to integrate a query-based verification tool into their existing system and QMaxUSE could be one of the key components to their next upgrade.

### 6.3. Answer to Research Question 3 (RQ3)

**Benchmark.** We use a benchmark from (Wu & Farrell 2021) to show the complexities and size of OCL invariants that QMaxUSE can handle. This benchmark consists of two parts.

[5] We plan to implement a technique for verifying OCL operational contracts in the next major release(Wu 2019; Wu. & Timoney. 2020).

[6] Team leader: huangxinx@sysucc.org.cn

| Name | OCL | | | | | Cons | Conf | Conc | $\geq 50$ |
|---|---|---|---|---|---|---|---|---|---|
| | Qtf | Col | Op | Nav | Oc | | | | |
| **EMF2CSP (González Pérez et al. 2012; Cabot et al. 2014)** | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✗ | ✗ |
| **OCL-Lite (Queralt et al. 2012b)** | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✗ | ✗ | ✗ |
| **CD2Alloy (Maoz et al. 2011)** | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ |
| **USE (Gogolla et al. 2018)** | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✗ | ✗ | ✗ |
| **ASMIG (Wu et al. 2013)** | ✓ | ✗ | ✓ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ |
| **MaxUSE (Wu 2017b; Wu & Farrell 2021)** | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | ✗ | ✓ |
| **QMaxUSE (Wu 2022)** | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ |

**Table 5** Comparison between QMaxUSE and other exisitng tools. Qtf=support quantifier, Col=support collection, Nav=support navigation, Oc=support operational contracts, Cons=consistency verification, Conf=pinpointing conflicts, Conc=support Concurrency, $\geq 50$ =more than 50 sample OCL invariants available.

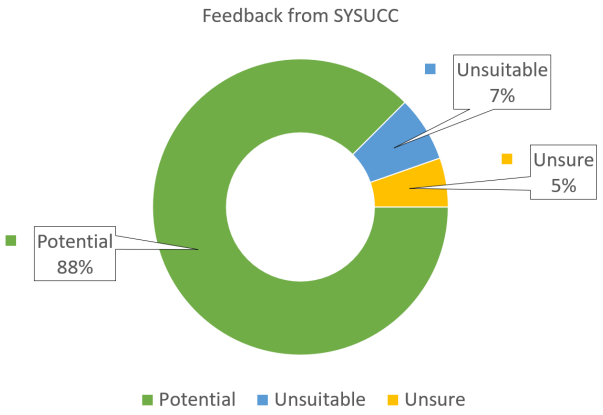



**Figure 7** A summary of feedback from SYSUCC. 88% of the team members believe QMaxUSE has potential to be applied to industrial problems. 7% of the team members think QMaxUSE is unsuitable for solving industrial problems. 5% of the team members are unsure about QMaxUSE.

The first part (Group A [7]) contains a list of models collected from (Gogolla et al. 2013) that was originally designed as a small benchmark for evaluating OCL invariants. The second part (Group B - Group E) contains a much larger and complex number of OCL invariants. We choose this benchmark [8] as it not only covers a wide range of OCL language features (nested quantifiers, inheritance, multiplicities, collections, navigation and a mixture of arithmetic/logic operators) but also a large number of *conflicted OCL invariants*.

**Experiment Setup.** We setup two experiments to effectively evaluate QMaxUSE's stability and performance. For the first experiment, we use MaxUSE to verify the full set of features of the models from our benchmark. MaxUSE is a tool that is able to find achievable features and pinpoint conflicting features without using our query language and concurrent algorithm. We then use QMaxUSE's query language and its concurrent algorithm to verify each model from our benchmark. For the second experiment, we evaluate QMaxUSE's concurrent verification algorithm on the models from our benchmark [9] but with a fixed number of threads. For both experiments, we use the Z3 SMT solver version 4.8.10 and run both tools on a machine that has a six 2.8GHz cores (Intel i5-9400F) with 16G memory.

**Results.** The complete benchmark and our both experimental results are shown in Table 6 and Table 7. Our results suggest that QMaxUSE is able to decompose OCL invariants into many different queries and all queries can be verified efficiently using a small number of threads. Compared to MaxUSE (without query and *Decompose* algorithm), our *Decompose* algorithm significantly reduces a large model into a number of smaller size of formulas. This boosts QMaxUSE's performance in verification and it can gain up to 30x efficiency improvement. For example, MaxUSE takes 131.23 seconds to verify $B3$ in Group B without using our query and *Decompose* techniques while QMaxUSE is able to finish its verification in just 4.6 seconds. This is particularly powerful and many large models that could not previously be verified now can be tackled by QMaxUSE. For example, $D4$ has about 101 invariants that cannot be verified within 20 minutes. However, QMaxUSE is able to break it into 56 smaller queries and verify them in about 8 seconds.

**Analysis.** QMaxUSE can significantly improve verification performance. This is mainly due to two reasons. (1) We directly extract unsat cores from the SMT solver (Z3) through its dedicated APIs. This is fundamentally different from our previous tool MaxUSE. The algorithms used in MaxUSE employ a two phase verification algorithm (Wu & Farrell 2021). At the first phase, it tries to satisfy as many OCL invariants as possible by reducing to a weighted MaxSMT problem. At the second phase, it pinpoints conflicting invariants by solving the set cover problem (Liffiton & Sakallah 2008). Though this algorithm is

[7] Group A contains one class diagram that uses a recursive OCL expression. Currently, QMaxUSE cannot handle recursive calls. Hence, this class diagram is removed from Group A.

[8] Currently, QMaxUSE supports OCL language features defined in this benchmark and skips other features. This is due to the capabilities of our first-order translator (Figure 4).

[9] We exclude Group A here since it contains only a small number of OCL invariants.

| | Name | Structure Size | | | | MaxUSE(sec) | QMaxUSE (sec) | |
|---|---|---|---|---|---|---|---|---|
| | | Invariants | Nodes | Quantifiers | Operators | Time | Queries | Time |
| Group A | A1 | 6 | 30 | 3 | 9 | 0.38 | 2 | 0.364 |
| | A2 | 7 | 52 | 8 | 1 | 0.148 | 1 | 0.087 |
| | A3 | 1 | 7 | 2 | 1 | 0.174 | 3 | 0.426 |
| | A4 | 8 | 73 | 7 | 18 | 0.204 | 3 | 0.241 |
| Group B | B1 | 27 | 150 | 10 | 30 | 4.528 | 23 | 2.022 |
| | B2 | 45 | 266 | 13 | 57 | 56.846 | 32 | 3.046 |
| | B3 | 68 | 430 | 9 | 111 | 131.23 | 42 | 4.604 |
| | B4 | 90 | 599 | 23 | 152 | 158.911 | 68 | 7.151 |
| | B5 | 136 | 925 | 44 | 228 | TO | 90 | 118.64 |
| Group C | C1 | 29 | 201 | 24 | 33 | 38.167 | 19 | 3.413 |
| | C2 | 43 | 279 | 28 | 51 | 91.319 | 33 | 5.954 |
| | C3 | 66 | 413 | 42 | 82 | 154.33 | 58 | 6.051 |
| | C4 | 98 | 698 | 69 | 137 | TO | 68 | 8.111 |
| | C5 | 156 | 1008 | 100 | 184 | TO | 99 | 114.41 |
| Group D | D1 | 22 | 174 | 23 | 36 | 4.373 | 18 | 1.575 |
| | D2 | 47 | 286 | 29 | 68 | 32.357 | 35 | 3.112 |
| | D3 | 61 | 324 | 23 | 72 | 27.026 | 46 | 5.083 |
| | D4 | 101 | 753 | 102 | 163 | TO | 56 | 8.211 |
| | D5 | 166 | 1143 | 131 | 225 | TO | 95 | 14.026 |
| Group E | E1 | 31 | 294 | 12 | 86 | 11.559 | 17 | 1.701 |
| | E2 | 39 | 452 | 18 | 135 | 42.255 | 20 | 2.779 |
| | E3 | 37 | 403 | 31 | 102 | 59.535 | 27 | 2.587 |
| | E4 | 105 | 985 | 56 | 246 | TO | 42 | 4.464 |
| | E5 | 167 | 1134 | 68 | 325 | TO | 45 | 3.653 |

**Table 6** Evaluation results. Invariants=number of OCL invariants, Nodes=size of invariant ASTs, Quantifiers=number of quantifiers, Operator=number of arithmetic and logic operators. TO= Timeout (20min), MaxUSE=QMaxUSE without query and concurrent verification support.

| Name | 8 Threads | 4 Threads | 2 Threads | 1 Thread |
|---|---|---|---|---|
| B1 | 2.145 | 2.174 | 2.986 | 3.957 |
| B2 | 3.093 | 3.230 | 3.648 | 4.429 |
| B3 | 4.654 | 4.856 | 5.290 | 5.790 |
| B4 | 7.160 | 7.172 | 7.804 | 8.787 |
| B5 | 118.541 | 118.860 | 118.922 | 122.580 |
| C1 | 3.425 | 3.454 | 3.691 | 4.406 |
| C2 | 5.840 | 5.846 | 5.978 | 6.309 |
| C3 | 6.125 | 6.200 | 6.374 | 6.381 |
| C4 | 8.113 | 8.293 | 8.513 | 10.563 |
| C5 | 114.460 | 115.41 | 116.391 | 118.831 |
| D1 | 1.572 | 1.572 | 1.592 | 1.602 |
| D2 | 3.112 | 3.112 | 3.242 | 3.506 |
| D3 | 5.075 | 5.083 | 5.632 | 6.903 |
| D4 | 8.221 | 8.245 | 8.347 | 9.906 |
| D5 | 14.045 | 14.383 | 14.983 | 15.890 |
| E1 | 1.700 | 1.720 | 1.723 | 1.820 |
| E2 | 2.740 | 2.774 | 3.778 | 4.890 |
| E3 | 2.565 | 2.580 | 2.599 | 3.420 |
| E4 | 4.428 | 4.774 | 5.778 | 6.890 |
| E5 | 2.565 | 3.080 | 3.899 | 5.420 |

**Table 7** Evaluation results form QMaxUSE with a fixed number of threads. The time unit here is seconds.

generic, it can be used for any solvers that do not support unsat core extraction. However, it can easily become very slow when the number of formulas increases. QMaxUSE operates differently from MaxUSE. It directly extracts unsat cores from the solver without implementing an extra layer on top of the solver. However, the limitation here is that a solver must support unsat core extraction. (2) Our results from Table 7 suggest that the *Decompose* algorithm in Section 4 is able to generate a number of much smaller size of formulas. These formulas can be verified efficiently using a small number of threads (even using a single thread). This can significantly improve QMaxUSE's performance and makes it possible to verify a large number of OCL invariants. We observe that the number of nested quantifiers in a formula ($\forall\ \exists$) can pose a significant challenge to the solver. Therefore, a verification engine that relies on the solvers may have the same challenge especially when the number of nested formulas dramatically increases. In a different manner, QMaxUSE is able to shift solving these large size of formulas to runs on a much smaller size of formulas. This yields a much better result and performance.

### 6.4. Strengths and Limitations

During the evaluation of QMaxUSE, we have identified the strengths and limitations of QMaxUSE, respectively.

**Strengths.** (1) Our query language can provide users with a way of incrementally verifying their UML class diagrams with OCL invariants by selecting different parts of their design. Each query issued by a user can be considered as a test case and this is particularly useful when users adopt test-driven development for their software development lifecycle. (2) Our concurrent verification algorithms can efficiently handle a large number and complex OCL invariants. This algorithm can not only verify the consistencies of a UML class diagram but also efficiently pinpoint conflicting OCL invariants. This is particularly powerful when the number of OCL invariants significantly increases.

**Limitations** (1) After discussion with the engineers from SYSUCC, we realised that many industrial problems possess many string constraints. Currently, QMaxUSE does not support string reasoning and skips an OCL invariant uses a string expression. Hence, we are working on improving QMaxUSE by integrating an efficient string solver. (2) Our verification algorithms rely on extracting unsat cores from an SMT Solver. However, not every SMT solver supports unsat core extraction. Furthermore, different SMT solvers might have their own strengths in solving a particular type of constraints. In the future, we plan to make a new solver-interface that allows users to choose the solver that has the best performance in a particular type of constraint solving. (3) Though our benchmark covers a wide range of OCL features including: nested quantifiers, navigations, (arithmetic/logical) operators and operations on collection data types. However, QMaxUSE does not support full range of OCL features such as closure operators and navigation (using self keyword). Hence, there is a gap between the full set of OCL features and current version of QMaxUSE.

## 7. Related Work

Recently, a significant number of approaches and techniques have been proposed to verify UML class diagrams (Büttner et al. 2012; Soeken et al. 2010; Wille et al. 2012; Balaban & Maraee 2013; Maraee & Balaban 2014; Milicevic et al. 2015). However, most of them suffer two main issues: (1) when a UML class diagram is inconsistent, they are unable to pinpoint conflicting OCL invariants. (2) when the number of OCL invariants increases, they do not scale well and often unable to progress (Cabot et al. 2014; Kuhlmann et al. 2011; Dania & Clavel 2016; Maoz et al. 2011). Among them, MaxUSE represents a new generation of verification tools in this area(Wu & Farrell 2021; Wu 2017b,a). It is able to verify a UML class diagram annotated with ranked OCL invariants and locate constraint conflicts. However, MaxUSE does not scale well when a large number of complex OCL invariants introduced. QMaxUSE advances the technique by implementing a query language and a new concurrent verification algorithm.

Approaches based on constraint solving techniques also have emerged (Büttner et al. 2012; Soeken et al. 2010; Wille et al. 2012; Dania & Clavel 2016; Wu et al. 2013; Kuhlmann et al. 2011; Wu 2016; Soeken et al. 2011). For example, Cabot et al.

use constraint programming (CP) to reprogram a UML class diagram with OCL invariants into a constraint satisfaction problem (CSP) that is solved later using constraint solvers (Cabot et al. 2009; González Pérez et al. 2012; Cabot et al. 2014). 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 (Büttner et al. 2012; Clavel et al. 2009).

Alloy as a model finder, is a popular tool that receives much attention in many areas including the Model Driven Engineering (MDE) community (Jackson 2002; Torlak & Jackson 2007). There has been much work on using Alloy to test/verify specifications of both semi-formal models and formal specifications (Perrouin et al. 2010; Gheyi et al. 2005; Milicevic et al. 2015). Since Alloy can be used to find model instances, research with Alloy has been highly active (Anastasakis et al. 2010; Garis et al. 2011; Kuhlmann et al. 2011; Kuhlmann & Gogolla 2012a,b). Most of this literature uses Alloy as a back-end reasoning engine to check consistencies of a UML class diagram. Among them, Anastasakis et al. focus on a transformation between UML class diagrams and Alloy's relational specification language (Anastasakis et al. 2007, 2010). In (Kuhlmann et al. 2011), Kuhlmann et al. integrate kodkod (Alloy's reasoning engine) into the USE modeling tool and translate OCL collection data types into Alloy (Kuhlmann & Gogolla 2012b). The main advantage of using Alloy is that it possesses a dedicated algorithm for finding minimal conflicts in the specification (Torlak et al. 2008). Hence, users are not required to have knowledge about SAT encoding details. However, Alloy currently does not support concurrent verification. Thus, Alloy cannot deal with large numbers of complex OCL invariants. In particular, expressions involves a number of arithmetic operators. This is because these operators can cause a plain SAT-solver to suffer bit-blasting. Therefore, approaches using Alloy as a basis for a constraint solving engines are restricted by this functionality (Anastasakis et al. 2007; Kuhlmann et al. 2011; Maoz et al. 2011; Garis et al. 2011; Kuhlmann & Gogolla 2012a)

Graph-based approaches are also widely used in verifying the consistencies of a UML class diagram (Ehrig et al. 2009; Hoffmann & Minas 2010, 2011; Balaban & Maraee 2013; Maraee & Balaban 2014). 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 class diagram 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 (Winkelmann et al. 2008). The OCL constraints in this approach are restricted to equality, size and attribute operations. Others in this domain devise specific algorithms that generate consistent graphs. 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 (Balaban & Maraee 2013). The FiniteSat algorithm transforms a class diagram with multiplicity constraints into a linear inequality system. However, this algorithm does not support any OCL constraints. More recently, Semeráth et al. proposed a new graph solver that is capable of generating a much larger number of objects (Semeráth et al. 2018). Their approach utilises a combination of multiple advanced graph-based and SAT-solving techniques to achieve large-scale graphs generation.

Model transformation languages such as ATL usually process a large number of transformation rules that conform to the OCL standard. Hence, the slicing techniques introduced in this area is also relevant to our work. For example, Cheng et al. propose a slicing technique for ATL model transformations (Cheng & Tisi 2017, 2018). Their technique can decompose a set of transformation rules into multiple sub-goals and captures a more precise fault localization using static trace information. Cuadrodo et al propose a method for the static analysis of ATL model transformations (Cuadrado et al. 2017). This method can compute a set of small metamodel footprints of the transformation that later on can then be solved by a model finder. This can reduce model finding times and yields a very fast analysis.

Different slicing techniques have been applied to UML class diagrams and OCL invariants (Sun et al. 2013; Lano & Kolahdouz-Rahimi 2010; Shaikh et al. 2010). UML2CSP is a tool that can slice a UML class diagram into submodels that can be verified within ECLipSE Constraint Programming Systems (Cabot et al. 2007). However, UML2CSP cannot partition a model if a common class or attribute is used by several OCL constraints. This means that it provides no improvement to the verification time. Sun et al. introduce a slicing technique that is able to decompose a UML class diagram with OCL invariants into different model fragments (Sun et al. 2013). These model fragments can later be analysed separately using the Alloy analyzer (Chang 2007). Kevin et al. propose slicing techniques for UML models including class diagrams and state machines (Lano & Kolahdouz Rahimi 2011). Their approach focuses on reactive specifications and the set of class invariants are converted into a series of conjunctions of predicates.

By comparing these approaches, QMaxUSE fundamentally differs from these approaches by providing a query language and a new algorithm that is capable of improving the verification time significantly. This query language gives users freedom to choose the parts of a UML class diagram to be verified and incrementally help them to eliminate design flaws in their models. Our concurrent algorithm can help them quickly identify the conflicting constraints even when the number of OCL invariants increase significantly.

## 8. Conclusions & Future Work

In this paper, we have presented a query based approach to verifying UML class diagrams with a large number of OCL invariants. Our prototype tool QMaxUSE demonstrates its potential to be adapted by industry. We believe that our approach advances current verification techniques in two unique ways: (1) It allows users to incrementally test/verify their design via a query language. (2) A large number of OCL invariants now can be decomposed into individual queries that can be verified concurrently.

During the work described in this paper, we identify two

interesting research directions. We outline the possible approaches here. (1) Many industry-size problems rely on string constraints to express their business rules. QMaxUSE currently does not support string reasoning. We are investigating multiple string solvers and will assess their capabilities by evaluating them on a comprehensive OCL string benchmark. We plan to integrate string solvers into QMaxUSE in the near future. We are investigating multiple string solvers and will assess their capabilities by evaluating them on a comprehensive OCL string benchmark. (2) During our evaluation of QMaxUSE, we find that different SMT solvers has its own strengths and limitations in solving different types of constraints. To fully harness their performance, we plan to introduce an artificial intelligence (AI) based approach that will predict the performance of a SMT solver and then provide a recommendation (Healy et al. 2016). One possible way is to train set of constraint problems and associated solver performance times. A machine learning algorithm that would then identify the best solver to use when a new but similar set of OCL constraints is presented.


### Acknowledgments

We would like to thank the reviewers of this document template for their helpful comments and suggestions.


## References


Ali, S., Yue, T., Zohaib Iqbal, M., & Panesar-Walawege, R. K. (2014). Insights on the use of ocl in diverse industrial applications. In D. Amyot, P. Fonseca i Casas, & G. Mussbacher (Eds.), *System analysis and modeling: Models and reusability* (pp. 223–238). Cham: Springer International Publishing.

Anastasakis, K., Bordbar, B., Georg, G., & Ray, I. (2007). UML2Alloy: A challenging model transformation. In *International conference on model driven engineering languages and systems* (p. 436-450). Springer.

Anastasakis, K., Bordbar, B., Georg, G., & Ray, I. (2010). On challenges of model transformation from UML to Alloy. *Software and System Modeling*, *9*(1), 69-86.

Atkinson, C., & Kühne, T. (2003). Model-driven development: A metamodeling foundation. *IEEE Software*, *20*(5), 36-41.

Balaban, M., & Maraee, A. (2013). Finite Satisfiability of UML Class Diagrams with Constrained Class Hierarchy. *ACM Transactions on Software Engineering and Methodology*, *22*(3), 24:1–24:42.

Berardi, D., Calvanese, D., & Giacomo, G. D. (2005). Reasoning on UML class diagrams. *Artificial Intelligence*, *168*(1–2), 70-118.

Booch, G., Rumbaugh, J., & Jacobson, I. (2005). *The Unified Modeling Language User Guide The Second Edition*. Addison-Wesley Professional.

Büttner, F., Egea, M., & Cabot, J. (2012). On verifying ATL transformations using 'off-the-shelf' SMT solvers. In *International conference on model driven engineering languages and systems* (p. 432-448). Springer.

Cabot, J., Clarisó, R., & Riera, D. (2007). UMLtoCSP: a tool for the formal verification of UML/OCL models using constraint programming. In *22nd international conference on automated software engineering* (p. 547-548). Atlanta, GA: IEEE Computer Society.

Cabot, J., Clarisó, R., & Riera, D. (2009). Verifying UML/OCL operation contracts. In *International conference on integrated formal methods* (p. 40-55). Springer.

Cabot, J., Clarisó, R., & Riera, D. (2014). On the verification of UML/OCL class diagrams using constraint programming. *Journal of Systems and Software*, *93*, 1-23.

Cadoli, M., Calvanese, D., Giacomo, G., & Mancini, T. (2007). Finite model reasoning on UML class diagrams via constraint programming. In *Artificial intelligence and human-oriented computing* (p. 36-47). Springer.

Chang, F. (2007). *The Alloy Analyzer 4.0.* http://alloy.mit.edu/. Retrieved from http://alloy.mit.edu/

Cheng, Z., & Tisi, M. (2017). A deductive approach for fault localization in atl model transformations. In *Fundamental approaches to software engineering* (pp. 300–317). Springer.

Cheng, Z., & Tisi, M. (2018). Slicing ATL model transformations for scalable deductive verification and fault localization. *International Journal on Software Tools for Technology Transfer*, *20*(6), 645–663.

Clavel, M., Egea, M., & de Dios, M. A. G. (2009). Checking unsatisfiability for OCL constraints. *Electronic Communication of the European Association of Software Science and Technology*, *24*.

Cuadrado, J. S., Guerra, E., & de Lara, J. (2017). Static analysis of model transformations. *IEEE Transactions on Software Engineering*, *43*(9), 868-897.

Dania, C., & Clavel, M. (2016). 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* (p. 65-75). ACM.

De Moura, L., & Bjørner, N. (2008). Z3: an efficient SMT solver. In *International conference on tools and algorithms for the construction and analysis of systems* (p. 337-340). Springer.

Ehrig, K., Küster, J. M., & Taentzer, G. (2009). Generating instance models from meta models. *Software and Systems Modeling*, *8*(4), 479-500.

Garis, A., Cunha, A., & Riesco, D. (2011). Translating Alloy Specifications to UML Class Diagrams Annotated with OCL. In *International conference on software engineering and formal methods* (p. 221-236). Springer.

Garry, D., & Balfe, T. (2012). Experiences using OCL for business rules on financial messaging. In *Proceedings of the 12th workshop on OCL and textual modelling* (p. 65–66). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/2428516.2428529

Gheyi, R., Massoni, T., & Borba, P. (2005). A rigorous approach for proving model refactorings. In *International conference on automated software engineering* (pp. 372–375). ACM.

Gogolla, M., Büttner, F., & Cabot, J. (2013). Initiating a benchmark for UML and OCL analysis tools. In *International conference on tests and proofs* (pp. 115–132). Springer.

Gogolla, M., Hilken, F., & Doan, K. (2018). Achieving model quality through model validation, verification and exploration.

*Computer Languages, Systems and Structures*, *54*, 474–511.

González Pérez, C. A., Buettner, F., Clarisó, R., & Cabot, J. (2012). EMFtoCSP: A tool for the lightweight verification of EMF models. In *International workshop on formal methods in software engineering: Rigorous and agile approaches* (p. 44-50). IEEE.

Healy, A., Monahan, R., & Power, J. F. (2016, November 8). Predicting SMT solver performance for software verification. In *3rd workshop on formal integrated development environment* (Vol. 240, p. 20-37). Limassol, Cyprus.

Hoffmann, B., & Minas, M. (2010). Defining models - meta models versus graph grammars. *Electronic Communications of the EASST*, *29*, 1-14.

Hoffmann, B., & Minas, M. (2011). Generating instance graphs from class diagrams with adaptive star grammars. In *International workshop on graph computation models.* Electronic Communications of the EASST.

Iqbal, M. Z., Ali, S., Yue, T., & Briand, L. (2012). Experiences of applying uml/marte on three industrial projects. In R. B. France, J. Kazmeier, R. Breu, & C. Atkinson (Eds.), *Model driven engineering languages and systems* (pp. 642–658). Berlin, Heidelberg: Springer Berlin Heidelberg.

Jackson, D. (2002). Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering Methodologies*, *11*(2), 256-290.

Kuhlmann, M., & Gogolla, M. (2012a). From uml and ocl to relational logic and back. In *International conference on model driven engineering languages and systems* (p. 415-431). Springer.

Kuhlmann, M., & Gogolla, M. (2012b). Strengthening SAT-based validation of UML/OCL models by representing collections as relations. In *Modelling foundations and applications* (Vol. 7349, p. 32-48). Springer.

Kuhlmann, M., Hamann, L., & Gogolla, M. (2011). Extensive validation of OCL models by integrating SAT solving into USE. In *International conference on objects, models, components, patterns* (p. 290-306). Springer.

Lano, K., & Kolahdouz-Rahimi, S. (2010). Slicing of uml models using model transformations. In *Proceedings of the 13th international conference on model driven engineering languages and systems: Part ii* (p. 228–242). Berlin, Heidelberg: Springer-Verlag.

Lano, K., & Kolahdouz Rahimi, S. (2011, 05). Slicing techniques for uml models. *The Journal of Object Technology*, *10*. doi: 10.5381/jot.2011.10.1.a11

Liffiton, M. H., & Sakallah, K. A. (2008, Janurary). Algorithms for computing minimal unsatisfiable subsets of constraints. *J. Autom. Reason.*, *40*(1), 1-33.

Maoz, S., Ringert, J. O., & Rumpe, B. (2011). CD2Alloy: Class diagrams analysis using alloy revisited. In *International conference on model driven engineering languages and systems* (p. 592-607). Springer.

Maraee, A., & Balaban, M. (2007). Efficient reasoning about finite satisfiability of UML class diagrams with constrained generalization sets. In *3rd european conference model driven architecture* (p. 17-31). Springer.

Maraee, A., & Balaban, M. (2014). Removing redundancies and deducing equivalences in UML class diagrams. In *International conference model-driven engineering languages and systems* (p. 235-251). Springer.

Milicevic, A., Near, J. P., Kang, E., & Jackson, D. (2015). Alloy*: A general-purpose higher-order relational constraint solver. In *International conference on software engineering* (pp. 609–619). IEEE.

Perrouin, G., Sen, S., Klein, J., Baudry, B., & l. Traon, Y. (2010, April). Automated and scalable t-wise test case generation strategies for software product lines. In *International conference on software testing, verification and validation* (p. 459-468). IEEE.

Queralt, A., Artale, A., Calvanese, D., & Teniente, E. (2012a). OCL-Lite: Finite reasoning on UML/OCL conceptual schemas. *Data & Knowledge Engineering*, *73*, 1-22.

Queralt, A., Artale, A., Calvanese, D., & Teniente, E. (2012b). Ocl-lite: Finite reasoning on UML/OCL conceptual schemas. *Data & Knowledge Engineering*, *73*, 1 - 22.

Queralt, A., & Teniente, E. (2006). Reasoning on uml class diagrams with ocl constraints. In *Conceptual modeling* (pp. 497–512). Springer.

Semeráth, O., Nagy, A. S., & Varró, D. (2018). A graph solver for the automated generation of consistent domain-specific models. In *Proceedings of the 40th international conference on software engineering* (p. 969–980). Association for Computing Machinery.

Shaikh, A., Clarisó, R., Wiil, U. K., & Memon, N. (2010). Verification-driven slicing of uml/ocl models. In *Proceedings of the ieee/acm international conference on automated software engineering* (p. 185–194). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/1858996.1859038 doi: 10.1145/1858996.1859038

Soeken, M., Wille, R., & Drechsler, R. (2011, March). Verifying dynamic aspects of uml models. In *Design, automation test in europe* (p. 1-6). IEEE.

Soeken, M., Wille, R., Kuhlmann, M., Gogolla, M., & Drechsler, R. (2010). Verifying UML/OCL models using boolean satisfiability. In *Design, automation test in europe* (p. 1341-1344). IEEE.

Sun, W., France, R. B., & Ray, I. (2013). Contract-aware slicing of uml class models. In A. Moreira, B. Schätz, J. Gray, A. Vallecillo, & P. Clarke (Eds.), *Model-driven engineering languages and systems* (pp. 724–739). Berlin, Heidelberg: Springer Berlin Heidelberg.

Torlak, E., Chang, F. S.-H., & Jackson, D. (2008). Finding minimal unsatisfiable cores of declarative specifications. In *International symposium on formal methods* (p. 326-341). Springer.

Torlak, E., & Jackson, D. (2007). Kodkod: a relational model finder. In *International conference on tools and algorithms for the construction and analysis of systems* (p. 632-647). Springer.

Wille, R., Soeken, M., & Drechsler, R. (2012). Debugging of inconsistent UML/OCL models. In *Design, automation test in europe* (p. 1078-1083). IEEE.

Winkelmann, J., Taentzer, G., Ehrig, K., & Küster, J. M. (2008).

Translation of restricted OCL constraints into graph constraints for generating meta model instances by graph grammars. *Electronic Notes in Theoretical Computer Science*, *211*, 159-170.

Wu, H. (2016). Generating metamodel instances satisfying coverage criteria via SMT solving. In *International conference on model-driven engineering and software development* (p. 40-51). IEEE.

Wu, H. (2017a). Finding achievable features and constraint conflicts for inconsistent metamodels. In *European conference on modelling foundations and applications* (pp. 179–196). Springer.

Wu, H. (2017b). Maxuse: A tool for finding achievable constraints and conflicts for inconsistent UML class diagrams. In *Integrated formal methods* (pp. 348–356). Springer.

Wu, H. (2018). Step 0: An idea for automatic OCL benchmark generation. In *17th international workshop on ocl and textual modeling* (pp. 356–364). Springer.

Wu, H. (2019). Synthesising call sequences from OCL operational contracts. In *Acm/sigapp symposium on applied computing* (p. 1871-1873).

Wu, H. (2022). QMaxUSE: A query-based verification tool for UML class diagrams with OCL invariants. In *25th international conference on fundamental approaches to software engineering.* Munich, Germany: Springer.

Wu, H., & Farrell, M. (2021). A formal approach to finding inconsistencies in a metamodel. *Software and Systems Modeling*.

Wu, H., Monahan, R., & Power, J. F. (2013). Exploiting attributed type graphs to generate metamodel instances using an SMT solver. In *International symposium on theoretical aspects of software engineering* (p. 175-182). IEEE.

Wu., H., & Timoney., J. (2020). Verifying OCL operational contracts via SMT-based synthesising. In *Proceedings of the 8th international conference on model-driven engineering and software development - modelsward,* (p. 249-259). SciTePress. doi: 10.5220/0009340602490259

## About the author

**Hao Wu** is an assistant professor in the Department of Computer Science at 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 creating new verification tools and methods for verifying different kinds of models used in Model Driven Engineering.