Step 0: An Idea for Automatic OCL Benchmark Generation

Hao Wu

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

Abstract. Model Driven Engineering (MDE) is an important software development paradigm. Within this paradigm, models and constraints are essential components for expressing specifications of a software artefact. Object Constraint Language (OCL), a specification language that allows users to freely express constraints over different model features. However, one major issue is that the lack of OCL benchmarks makes difficult to evaluate existing and newly created OCL tools. In this paper, we present our initial idea about automatic OCL benchmark generation. The purpose of this paper is to show a developing idea rather than presenting a more formal and complete approach. Our idea is to use an OCL metamodel to sketch abstract syntax trees for OCL expressions, and solve generated typing constraints to produce the concrete OCL expressions. We illustrate this idea by using an example, discuss our work-in-progress and outline challenges to be tackled in the future.

1 Introduction & Related Work

Object Constraint Language (OCL), as a specification language in Model Driven Engineering (MDE), is *formally* used for writing rules that are not expressible by using models [1]. It plays a central role in many model-based engineering domains such as language engineering, model transformation and business process modelling. One particular example is ATL, a model transformation language that is built on top of OCL and it allows users to specify precise transformation rules for a set of model features. On the other hand, users can use OCL for different purposes including writing constraints/invariants for specific entities, specifying pre/post conditions over operations or methods and running queries over a set of features.

Recently, many approaches and techniques have been proposed for analysing or verifying models annotated with OCL [2,3,4,5,6,7,8,9,10,11,12,13]. These approaches either provide comprehensive case studies or tool support [6,14,15,16] for analysing OCL constraints. However, a major issue is the lack of OCL benchmark. This is difficult for users to evaluate or choose suitable OCL tools for their own projects. This issue has recently been addressed by Gogolla and Cabot [17] [18]. Forming a collection of OCL benchmarks is necessary for OCL communities. Typically, there are two ways of forming such collections: (1) Extensively

collecting existing models that are annotated with OCL constraints from different locations such as code repositories and modelling zoos. (2) Automatically generating a collection of OCL constraints with respect to user's requirements. For example, users may be interested in evaluating scalability of their own tools. Thus, they need a large number of OCL expressions. Further, users may also focus on evaluating a particular aspect of a tool such as conflict detection. In this scenario, it would be very useful to automatically generate a large number of conflicted OCL expressions.

In this paper, we propose an idea of automatic OCL benchmark generation. We consider this idea as a complement to the idea of forming a benchmark via manually collecting existing models annotated with OCL. By exploiting this idea, users could create customised benchmarks to accommodate their own purposes such as generating property-specific OCL expressions.

2 The Proposed Idea

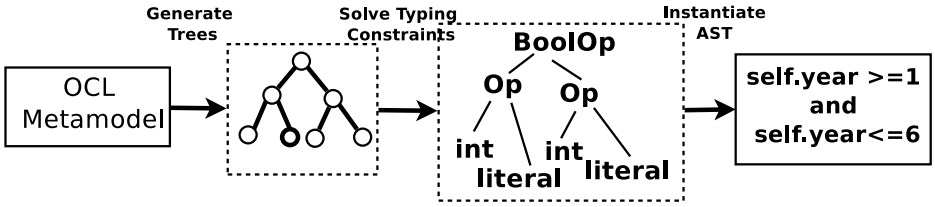

Fig. 1. The overview of an idea for generating a OCL benchmark.

Our idea for automatic generating OCL benchmark is visualised in Figure 1. Given a number of OCL constraints to be generated, users first define the properties for each OCL constraint. For example, a property call with a logic operator over an attribute. Here, we consider these properties are described in a standard OCL metamodel [1]. Second, we use a tree generator to generate the shape of an abstract syntax tree (AST) for each OCL constraint. This tree generator consults both the OCL metamodel and OCL concrete syntax to produce the ideal size of an AST, and generates a set of typing constraints for each AST. These constraints restrict possible types on each node in an AST. We then use an SMT solver to solve these constraints to derive a precise type for each node. Finally, we traverse the AST and instantiate each node with a concrete value. To form a OCL benchmark, we repeat these steps until the number of OCL constraints a user asked for is met.

2.1 An Example

In this section, we describe a scenario to illustrate our idea of automatic OCL benchmark generation. This scenario is based on our recent experience in evaluating a newly created OCL tool [19].

Figure 2 shows a UML class diagram that captures a relationship between a doctor and accident & emergency department in a hospital. Now consider a scenario where a user has already designed a tool for verifying OCL constraints, and would like to evaluate the performance and scalability of this tool on the OCL logical expressions with the model shown in Figure 2. In this case, existing collected OCL examples such as those are in [17] and [18] are no longer suitable for this scenario since they use different models and contain less number of constraints. Typically, measuring the performance and scalability of a tool involves running against a large number of OCL constraints. Further, this user requires a specific criteria that models must contain a large number of expressions using logical operators. Therefore, it would be very useful to generate a customised OCL benchmark for this specific scenario.

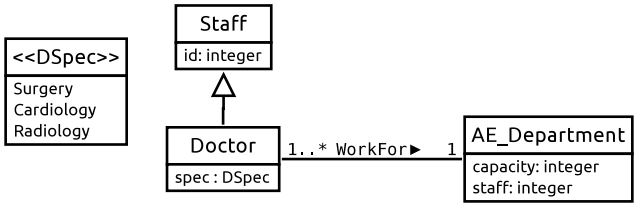

Fig. 2. A UML class diagram that represents a relationship between a doctor and Accident & Emergency department in a hospital.

To generate OCL logical expressions for this model, we first allow users to specify a type for each OCL constraint to be generated. To ensure the chosen types are valid, we use the standard OCL metamodel as a reference. For example, a user may select a property constraint for id attribute defined in the $Staff$ class. The property call of an OCL constraint corresponds to the $PropertyCallExp$ in the OCL metamodel that is shown in Figure 3. Note that a user may select the same constraint type for multiple model features. For the reason of simplicity, we assume that users only choose a constraint type involving a single model feature.

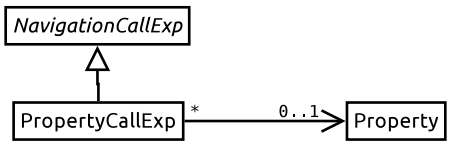

Fig. 3. A part of an OCL metamodel representing the relationship between two classes: $PropertyCallExp$ and $Property$.

Once the type of an OCL constraint has been fixed, we then use a tree generator to sketch the shape of an abstract syntax tree based on consulting

the OCL concrete syntax. At this stage, users may specify a particular type expression and tree size. For example, a user may select a binary expression for a property constraint over the attribute *id*. The tree generator then tries to generate a tree that has the specified size. However, the size may vary and depends on OCL concrete syntax. For example, Figure 4 shows an example of a generated abstract syntax tree for a binary expression. This tree has a size of 9, the root R produces two other binary expressions: $n1$ and $n2$.

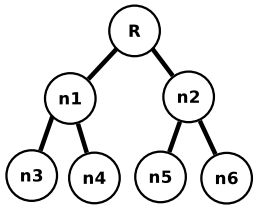

Fig. 4. An abstract syntax tree for a binary expression.

Now, we have the shape of an AST and the goal here is to work out correct types. More importantly, we need to ensure the type information preserved in an AST is consistent. For example, two boolean expressions cannot be connected by an arithmetic operator such as $+$ and $-$. In order to work out type information for each node, we generate a set of typing constraints for an AST and solve these constraints by using an SMT solver. To illustrate these typing constraints, we use Figure 4 as an example.

Assume the AST in Figure 4 represents a binary expression that captures an OCL property constraint for the attribute *id* in the class *Doctor* from Figure 2. Since this tree represents a binary expression, the root R must be a binary operator such as $>$ or *and*. Node $n1$ and $n2$ could be another two OCL expressions containing two children nodes respectively. One of the possible kinds of expressions is that $n1$ and $n2$ are two binary expressions as well. For the reason of simplicity, let us assume that this is the case. If $n1$ is a binary expression over *id*, then either $n3$ or $n4$ must be the attribute *id*[1]. Similarly, this is the same for $n5$ and $n6$.

Thus, we now can generate the following typing constraints for the AST in Figure 4.

$$(R \in OP_l) \wedge (n1 \in OP_c) \wedge (n2 \in OP_c) \wedge$$
$$\big(T(n3) = INT\big) \oplus \big(T(n3) = INT\_LITERAL\big) \wedge$$
$$\big(T(n4) = INT\big) \oplus \big(T(n4) = INT\_LITERAL\big) \wedge$$
$$\big(T(n5) = INT\big) \oplus \big(T(n5) = INT\_LITERAL\big) \wedge$$
$$\big(T(n6) = INT\big) \oplus \big(T(n6) = INT\_LITERAL\big) \wedge$$

Here, T is a function that returns a particular OCL type. Sets OP_l and OP_c represent all possible binary operators. For the sub-tree that contains nodes $n3$ and $n4$, exactly one of the nodes has an INT type. This is because the attribute

[1] In a more complex scenario, either $n3$ or $n4$ could also be an integer or an attribute.

id is an integer type. Since we consider a scenario that a constraint over a single attribute, the other node must be an integer literal ($INT\_LITERAL$) type[2]. Since each OCL constraint is a boolean expression and the tree represents a binary expression, R must be a logical operator. This implies that nodes $n1$ and $n2$ must be the operators that apply to two integer types and return a boolean type. For example, comparison operators: $>$ and $<$. Hence, we now can define the following operators for OP_l and OP_c.

$OP_l = \{and,\ or,\ xor,\ implies\}$

$OP_c = \{>,\ >=,\ <,\ <=,\ <>,\ =\}$

To generate constraints for OP_l and OP_c, we use an integer vairable to encode each operator and constrain this integer vairable to cover all possibilities. We then use an SMT solver to solve generated typing constraints and interpret the successful assignment for each node in the AST [20]. For example, Figure 5 (a) shows an example of solved type constraints for the AST in Figure 4.

Finally, we instantiate an AST with concrete values. Currently, we use a random value generation for each OCL literal type *string*,*int* and *boolean*. In this example, we use attribute *id* for each INT and randomly choose two integers for both $INT\_LITERAL$. The final resulting OCL constraint for the attribute *id* in the class *Doctor* is shown in Figure 5 (b).

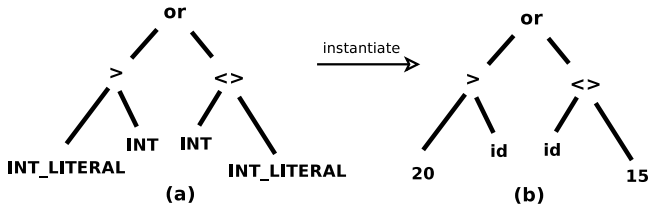

Fig. 5. (a) An example of solved typing constraints. (b) An abstract syntax tree with concrete values.

3 Work In Progress

We have implemented this idea into a prototype tool: OCLGen. We use OCLGen in our most recent work for generating a customised OCL benchmark to evaluate a technique for finding achievable features and OCL constraint conflicts[19,31]. OCLGen uses the examples presented by Gogolla and Cabot as candidate models and further generates a much larger number OCL constraints based on the calculated configuration [17]. The configuration contains a set of different parameters including number of the quantifiers, logical operators and navigations. These generated OCL constraints cover a variety of features such as constraints over multiple inheritances, the nested quantified OCL expressions and random constraint conflicts. At the moment, OCLGen is able to handle the generation of

[2] In a multiple attributes scenario, the node could be either an integer literal or another integer type attribute

simple binary and quantified OCL expressions containing arithmetic, navigation and logical operators. [3].

4 Challenges & Future Work

Though we have a working prototype for automatically generating OCL benchmarks, there are quite a few much more challenging problems remain.

1. Choosing/Designing an appropriate domain-specific language for describing benchmarks. Formally, users would be able to use a well-defined language to describe the kinds of benchmarks to be generated. For example, allowing users to quantify the number of operators in an OCL expression or specify the type of constraints to be generated such as navigations. Recently, a large number of OCL analysis and verification tools have been developed [21,22,16,6]. However, not many of them evaluated their tools on a large number of inconsistent OCL constraints. The challenge here is that this language not only allows users to specify valid number of OCL constraints to be generated but also constraints cause inconsistencies. The generated benchmarks thus can be used for the purpose of evaluating the soundness of an OCL analysis tool.
2. Measuring the generated computational complexity of OCL benchmarks using a set of metrics. Users may use different or the same OCL benchmarks for evaluating existing, or their own OCL tools for different purposes. In this context, a set of suitable metrics for a benchmark is necessary. Those metrics can be used as a standard way of measuring the computational complexity of an OCL benchmark so that researchers and users in the community could have a clear idea of what tools are capable of. Even if the evaluation is not performed on the same benchmark [23]. For example, the metrics may include the measurement of the number of OCL data types, the maximum/minimum (AST) size of generated OCL expressions, the depths of quantifiers, etc. Further, a much more challenging problem here is that to automatically generate a benchmark meeting those metrics so that users can use it for focusing on a particular aspect of an evaluation.
3. Generating OCL benchmarks efficiently and effectively. Typically, the generation process should be completed within a reasonable time frame. As it can be seen from the example in Section 2.1, the shape of an AST and its type information can be naturally and formally tackled by constraints. The properties of an OCL expression such as the number of quantifiers can also be expressed as SAT/SMT constraints. The use of constraint solvers (SAT/SMT) have been proven to be successful in many domains [24,25,7,26,27]. However, one problem of those solvers is that they usually do not scale very well. Based on our recent experience, we discover that sometimes those solvers may lose accuracy when the problem size is too big [19]. This is probably caused by the heuristic algorithms used within solvers. For this reason, the predication of how those solvers' will performance on a particular problem could be

[3] The fully generated benchmark is available at
`https://github.com/classicwuhao/maxuse/tree/master/maxuse_examples/benchmark`

helpful to tell users what to expect [28]. Additionally, a benchmark formed by a mixture of manually created examples with generated ones could be a practical way for determining where a numerous number of OCL constraints needed.

In this paper, we have presented our initial idea of automatically generating OCL benchmark by producing skeletons of OCL abstract syntax trees based on an OCL metamodel and solving generated typing constraints for each AST. The experience of using our prototype tool OCLGen is the very first step towards proposing a complete framework for automatic OCL benchmark generation.

In the long term, we plan to tackle the above challenges individually and continue extending our work in OCLGen. This involves investigating the design of a domain-specific language for generating metrics-oriented OCL benchmarks. Though we have done preliminary work on generating graph-oriented instances, OCL constraint generation is much more challenging since we need to take many aspects into account such as tree shapes and typing constraints [29,30]. Further, we will also enhance our tree generator to generate more complex structures such as queries over a collection data type. Our ultimate goal is to solve these challenges listed above and build a framework for automatically generating customised OCL benchmarks that can be used for evaluating OCL analysis and verification tools to accommodate different user requirements.

References

1. Object Management Group: Object Constraint Language Version 2.4 (2014)
2. 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)
3. Maraee, A., Balaban, M.: Efficient reasoning about finite satisfiability of UML class diagrams with constrained generalization sets. In: 3rd ECMDA, Springer (2007) 17–31
4. Brucker, A.D., Wolff, B.: HOL-OCL – A Formal Proof Environment for UML/OCL. In: The 11th FASE, Springer (2008) 97–100
5. 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. ENTCS **115** (2005) 39–47
6. Clavel, M., Egea, M., de Dios, M.A.G.: Checking unsatisfiability for OCL constraints. ECEASST **24** (2009)
7. Büttner, F., Egea, M., Cabot, J.: On verifying ATL transformations using 'off-the-shelf' SMT solvers. In: 15th MoDELS. (2012) 432–448
8. Soeken, M., Wille, R., Kuhlmann, M., Gogolla, M., Drechsler, R.: Verifying UML/OCL models using boolean satisfiability. In: DATE, Dresden, Germany (2010) 1341–1344
9. Queralt, A., Artale, A., Calvanese, D., Teniente, E.: Ocl-lite: Finite reasoning on UML/OCL conceptual schemas. Data & Knowledge Engineering **73** (2012) 1 – 22
10. Dania, C., Clavel, M.: Ocl2fol+: Coping with undefinedness. In: OCL@MoDELS. (2013)

11. Semeráth, O., Vörös, A., Varró, D.: Iterative and incremental model generation by logic solvers. In: 19th FASE, Springer (2016) 87–103
12. 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
13. 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
14. Soeken, M., Wille, R., Drechsler, R.: Encoding OCL data types for SAT-based verification of UML/OCL models. In: 5th TAP, Zurich, Switzerland, Springer (2011) 152–170
15. 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)
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. Gogolla, M., Büttner, F., Cabot, J. In: Initiating a Benchmark for UML and OCL Analysis Tools. Springer (2013) 115–132
18. Gogolla, M., Cabot, J. In: Continuing a Benchmark for UML and OCL Design and Analysis Tools. Springer (2016) 289–302
19. Wu, H.: Finding achievable features and constraint conflicts for inconsistent metamodels. In: 13th ECMFA. (2017)
20. De Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: 14th TACAS, Budapest, Hungary, Springer (2008) 337–340
21. Wille, R., Soeken, M., Drechsler, R.: Debugging of inconsistent UML/OCL models. In: 2012 DATE. (2012) 1078–1083
22. Balaban, M., Maraee, A.: Finite Satisfiability of UML Class Diagrams with Constrained Class Hierarchy. ACM Transacation on SEM. **22**(3) (2013) 24:1–24:42
23. Cabot, J., Teniente, E.: A metric for measuring the complexity of OCL expressions. In: Model Size Metrics Workshop@MODELS 2006.
24. Cadar, C., Dunbar, D., Engler, D.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: 8th USENIX Conference on Operating Systems Design and Implementation. (2008) 209–224
25. Peleska, J., Vorobev, E., Lapschies, F. In: Automated Test Case Generation with SMT-Solving and Abstract Interpretation. Springer (2011) 298–312
26. Gulwani, S., Jha, S., Tiwari, A., Venkatesan, R.: Synthesis of loop-free programs. In: The 32nd PLDI, ACM (2011) 62–73
27. Phothilimthana, P.M., Thakur, A., Bodik, R., Dhurjati, D.: Greenthumb: Superoptimizer construction framework. In: 25th CC, ACM (2016) 261–262
28. Healy, A., Monahan, R., Power, J.F.: Predicting SMT solver performance for software verification. In: 3rd Workshop on FIDE. (2016) 20–37
29. Wu, H.: Generating metamodel instances satisfying coverage criteria via SMT solving. In: The 4th MODELSWARD. (2016) 40–51
30. Wu, H.: An SMT-based Approach for Generating Coverage Oriented Metamodel Instances In: IJISMD **7** (2016), 23–50
31. Wu, H.: MaxUSE: A Tool for Finding Achievable Constraints and Conflicts for Inconsistent UML Class Diagrams. In: 13th integrated Formal Methods