A Systematic Approach to Generate Diverse Instantiations for Conceptual Schemas*

Loli Burgueño[1,2], Jordi Cabot[3], Robert Clarisó[1], and Martin Gogolla[4]

[1] Universitat Oberta de Catalunya, Barcelona, Spain
[2] Institut List, CEA, Université Paris-Saclay, France
[3] ICREA, Barcelona, Spain
[4] University of Bremen, Bremen, Germany
lburguenoc@uoc.edu, jordi.cabot@icrea.cat, rclariso@uoc.edu, gogolla@uni-bremen.de

Abstract. Generating valid instantiations for a conceptual schema is instrumental in ensuring its quality by means of verification, validation or testing. This problem becomes even more challenging when we also require that the computed instantiations exhibit significant differences among them, i.e., they are diverse. In this work, we propose an *automatic* method that guarantees synthesizing a diverse set of instantiations from a conceptual schema by combining model finders, classifying terms and constraint strengthening techniques. This technique has been implemented in the USE tool for UML/OCL.

Keywords: Methodologies and tools for conceptual design · Quality of conceptual models · Integrity constraints

1 Introduction

Verification, validation and testing are different mechanisms to ensure the quality of a conceptual schema. These approaches typically require the same resource: creating one or more instantiations of the conceptual schema. With "instantiation" we refer to an example for an information base of a conceptual schema [16]. These instantiations can be used as *illustrations* to better understand the model, to explain its behavior or to simulate it; as *counterexamples* that describe invalid configurations; and as *test cases* to capture scenarios that should be checked.

A key property of any set of instantiations to be used in a quality assurance process is its *diversity*. That is, the instantiations in the set should cover a broad spectrum of different configurations and scenarios. Otherwise, relevant corner cases might be missed, causing faults and/or wrong conclusions.

Asking a domain expert to create instantiations manually can be very time-consuming and is usually not feasible. Instead, dedicated tools called *model finders* [12, 18] can be used to automatically compute (valid) instantiations of a conceptual schema that satisfy all its integrity constraints. Model finders rely on different techniques like constraint solvers, theorem provers or search algorithms

* This work is partially funded by the H2020 ECSEL Joint Undertaking Project "MegaM@Rt2: MegaModelling at Runtime" (737494) and the Spanish Research Project TIN2016-75944-R.

to perform this computation [12]. While model finders automate the generation process, they do not guarantee diverse solutions.

One approach that helps model finders generate a diverse output is called *classifying terms* (CT) [14]. Classifying terms are properties that can be used to partition the solution space. Intuitively, a CT is an expression or property defined in such a way that two instantiations yielding different values for the classifying term will be (very) dissimilar. The partitions induced by classifying terms can guide the model finder and direct it to select canonical representatives from each partition, rather than arbitrary instantiations. Thus, a proper choice of classifying terms ensures a good partition and, thus, model diversity. Nevertheless, proposing suitable classifying terms requires domain knowledge. Thus, it is not trivial to automate and requires the participation of a domain expert.

To overcome this issue, this work proposes a method for automatically generating relevant classifying terms for a given conceptual schema. The approach arises from the observation that classifying terms are typically related to the integrity constraints in the schema. Therefore, we propose to *mutate* the schema's integrity constraints in a structured way in order to generate classifying terms. In particular, we extend and adapt the work on *constraint strengthening* [6] to produce these mutants of interest. Our approach, combining constraint strengthening, classifying terms and model finders, enables the automatic generation of diverse instantiations from a conceptual schema. This result is useful in many areas of conceptual modeling beyond information systems. To describe the method, and without loss of generality, we consider conceptual schemas expressed as UML class diagrams enriched with OCL constraints to describe integrity constraints.

The paper is structured as follows. Section 2 discusses the state of the art. Section 3 presents our proposed method for synthesizing classifying terms and Section 4 its implementation. Section 5 discusses the advantages and shortcomings of this method. Finally, Section 6 concludes and discusses future work.

2 State of the art

Some works on general purpose satisfiability solvers are focused on *random sampling* [7, 5, 8]), *i.e.*, finding diverse satisfying assignments to boolean formulas.

In the specific context of model finders, there are many approaches for finding valid instances for a model [12] but only a few consider diversity. Some strategies that have been used are *symmetry breaking* [18], *distance metrics* [9], *abstract graph shapes* [17] or *random restarts* [2]. One of these approaches lets the designer control diversity by defining *classifying terms* [14]: relevant boolean or integer expressions that partition the set of solutions by exhibiting different results for instantiations that are dissimilar.

Testing methods also rely on model finders to synthesize test cases [1, 3]. This process may require *mutating* constraints, selecting edit operators randomly from a predefined catalog. Instead, in this paper we modify constraints in a structured way [6] in order to *strengthen* them. Besides, we explore the (complex) partitions defined by classifying terms instead of simply defining ranges of "meaningful values" for inputs as in [10, 15].

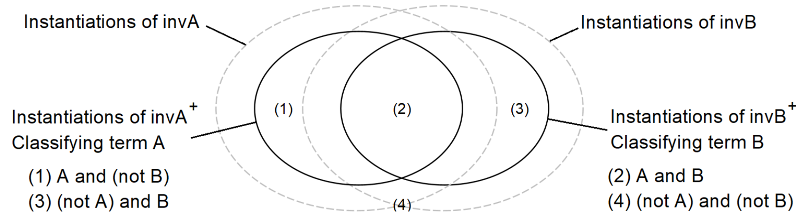

Fig. 1. Diverse instantiations through two Boolean classifying terms.

3 Our Generative Method

Our method adapts and combines techniques from two previous works: classifying terms (CT) [14] and the strengthening of integrity constraints [6].

CTs are employed to explore the set of possible correct or incorrect instantiations of a conceptual schema in order to obtain a few diverse instantiations instead of many similar ones. The approach enables the generation of instantiations that satisfy all constraints (positive test cases) as well as instantiations in which some constraints fail (negative test cases). However, until now, CTs had to be manually written by the developer, which limited the usability of the approach. To address this limitation, this work uses the existing integrity constraints in the model as an input and strengthen them (*i.e.*, it mutates them to generate a more restrictive version of the constraint) for obtaining *meaningful* CTs. That is, CTs that generate interesting equivalence classes that can be then taken as input for the generation of diverse instantiations in which border cases become clear. For example, one instantiation where a particular constraint holds and another one where the same constraint fails.

Figure 1 explains the basic idea behind our new approach. As an example, we consider two invariants invA and invB and assume two strengthened versions of them invA$^+$ and invB$^+$ have been identified. By considering the strengthened versions as CTs, one will ideally construct four model equivalence classes and obtain four diverse instantiations that show different behavior with respect to the two CTs. The next sections illustrate each step of this process in more detail using a running example.

3.1 Running example

As a running example throughout the paper we use the simple conceptual schema depicted in Figure 2, representing a simplified cloud provisioning model. Different cloud providers offer a number of *CloudService*s to the potential *Customer*s who put *Order*s based on their data volume needs. To ensure the integrity of the provisioning, a number of constraints are defined on top of the schema. For instance, we check that orders must be within the Customer budget (constraint `orderWithinBudget`) or that premium customers have at least one order with a data volume higher or equals to 5. Due to space constraints we only show three invariants below, the rest are available in our Git repository [4].

```
context Customer inv orderWithinBudget:
  self.ord->forAll(o |
    self.budget >= o.dataVol*o.serv.unitPrice)
context cs1,cs2:CloudService inv uniqueProviderMaxDataVol:
  cs1<>cs2 implies
  cs1.provider<>cs2.provider or cs1.maxDataVol<>cs2.maxDataVol
context Customer inv minimumDataVolCompany:
    self.premium implies self.ord->exists(o | o.dataVol>=5)
```

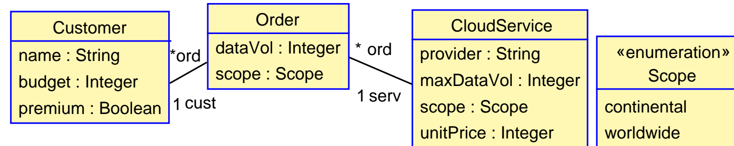

Fig. 2. A simple cloud provisioning schema

3.2 Derivation of classifying terms via constraint strengthening

Mutation is a technique used in the context of software testing. This process starts from a software artifact, usually a program or function, and systematically introduces small changes to produce new versions of the artifact called *mutants*. These syntactic changes, called *mutation operators*, are intended to mimic frequent developer errors. Then, it is possible to check whether a test suite is capable of detecting that the original artifacts and the mutants do not have the same behavior.

In the context of OCL integrity constraints, strengthening [6] is a method for *structured* mutation. By construction, strengthening guarantees that any mutant it produces is more restrictive than the original OCL constraint, taking into account the semantics of OCL (including OCL's 4-valued logic [13], which considers invalid and undefined values). This is achieved by ensuring that this property holds for each mutation operator.

As an example, below we show sample strengthening candidates (noted using the $^+$ symbol) applied to two expressions including the boolean operator *or* (left) and the relational operator `>=` (right). Note that, when the subexpressions are boolean, potential strengthenings may require to strengthen some of its subexpressions. A complete list of candidate strengthenings can be found in [6].

```
[exp1 or exp2]+  →  exp1 and exp2        [exp1 >= exp2]+  →  exp1 > exp2
                    exp1                                     exp1 = exp2
                    [exp1]+ or exp2                          exp1 > exp2 + 1
```

Strengthening was originally proposed as a way to suggest fixes for integrity constraints that were found to be too lax. Here, we adapt this method to generate classifying terms for a conceptual model. Notice that we are not interested in classifying terms that are more lax than the integrity constraints: if an instantiation does not satisfy an integrity constraint, it will be discarded as invalid. On the other hand, stronger versions of integrity constraints will produce valid

instances, which is exactly what we need and the reason why we use invariant strengthening to generate classifying terms.

A first adaptation is that classifying terms are not restricted by a context type like OCL invariants. Thus, we have to rewrite the OCL constraints to provide a meaning for the "self" object via `allInstances`. For instance, the invariant `orderWithinBudget` has to be rewritten as:

```
Customer.allInstances->forAll(c |
  c.ord->forAll(o | c.budget >= o.dataVol*o.serv.unitPrice))
```

Once the invariants are rewritten, we apply the strengthenings to obtain the classifying terms. As an example, we show below the resulting classifying term after applying three strengthenings to the three invariants presented above:

```
-- budget: strengthening '>=' -> '='
Customer.allInstances()->forAll(c | c.ord->forAll(o |
  c.budget = o.dataVol*o.serv.unitPrice))
-- uniqueness: strengthening 'or' -> 'and'
CloudService.allInstances()->forAll(cs1,cs2 | cs1<>cs2 implies
  cs1.provider<>cs2.provider and cs1.maxDataVol<>cs2.maxDataVol))
-- minimum: strengthening 'A implies B' -> 'B'
Customer.allInstances()->forAll(c| c.ord->forAll(o | o.dataVol >= 5))
```

3.3 Constructing diverse instantiations

As stated before and in [14], a classifying term is a closed OCL query expression that computes a `Boolean` or an `Integer` value, i.e., an OCL expression without free variables that, when evaluated in an instantiation, gives a `Boolean` or an `Integer` result. Given a collection of classifying terms $CT_1, \dots, CT_n$ for a conceptual model, the model finder internally operates as follows:

1. Compute an instantiation of the model respecting the stated OCL invariants.
2. Evaluate the classifying terms in the current instantiation (values $v_1, \dots, v_n$).
3. Internally add a new constraint forbidding that the classifying terms take the values of the found instantiation: $(CT_1 <> v_1)$ *or* ... *or* $(CT_n <> v_n)$.
4. Repeat the process until no more instantiations can be found.

In the running example, the three constructed Boolean classifying terms `budget`, `uniqueness` and `minimum` will yield eight (2^3) instantiations where in a single instantiation each classifying term is either `false` or `true`. Figure 3 shows two of the eight instantiations and the result of evaluating the three classifying terms in each instantiation.

4 Tool support

Our method has been implemented inside the USE tool [11]. USE is implemented in Java and provides, among other features: a GUI; packages to load, modify and inspect models and instantiations; and commands to invoke the KodKod relational solver [18] for model finding.

The *input* of our tool is a UML class diagram annotated with OCL invariants. This model is specified using the textual format employed by USE (`.use`). The

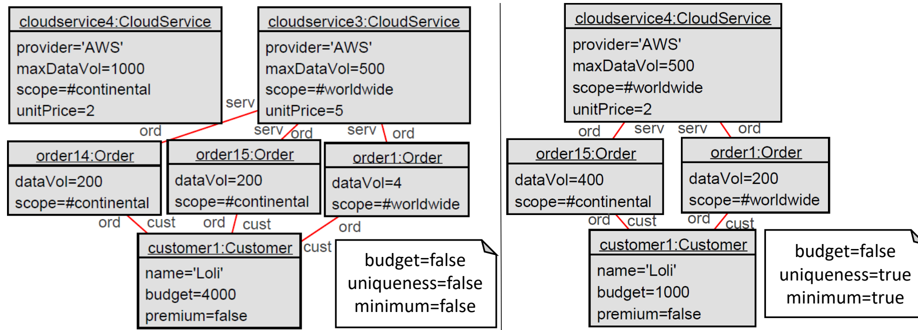

Fig. 3. Two generated instantiations for the Cloud Provisioning example

output is a set of diverse instantiations of the model, which can be visualized as object models within USE. Our implementation is divided into three steps:

1. *Candidate generation:* First, we strengthen each OCL integrity constraint by means of a post-order traversal of its abstract syntax tree (AST). For each boolean expression in the AST, we generate the candidate strenghtenings by combining the strenghtenings of its subexpressions (see Section 3.2). The list of candidates for the root of the AST is the list of potential classifying terms for that invariant.
2. *Candidate selection:* Next, we select the subset of candidates to be used as classifying terms to compute instantiations. This choice can be performed in different ways: randomly; using heuristics (*e.g.* choosing the classifying terms that involve the more constrained elements in the model); or with the help of a domain expert (note that choosing classifying terms from a list of candidates requires much less effort than proposing them).
3. *Computation of instantiations:* Finally, we provide the classifying terms to the model finder, which uses them to find representative instantiations in each of the equivalence classes induced by the partition. This process is automated within USE using the command *mv -scrollingAllCT <properties_file>*, which receives as parameters the properties file in which the verification bounds are stated; asks for the classifying terms; and invokes the Kodkod model finder to generate instantiations.

Our running example has been analyzed using this tool implementation, which is available for download from [4].

5 Discussion

Correctness and completeness. The classifying terms are always added on top of the existing integrity constraints in the schema. Therefore, any solution obtained using classifying terms satisfies all constraints. This guarantees the correctness of any generated instantiation.

Regarding the expressiveness of integrity constraints, our method supports all features of OCL 2.4 except for ordered collections (Sequence, OrderedSet) or recursively defined queries. Without loss of generality, we have focused on the generation of boolean classifying terms but a similar process could be applied for the generation of integer classifying terms.

Performance. Computing the classifying terms adds a performance overhead to the solution generation process, but it is negligible. The classifying term generation time was 151 milliseconds for our running example, which was 1-2 orders of magnitude faster than the time it took to compute a single valid instantiation.

Given that this second task (computing the instantiation) is the bottleneck, any approach aiming to reduce the number of times we need to trigger the generation of a new instantiation to ensure diversity in the result set[5], will significantly reduce the overall computing time.

Heuristics for the selection of classifying terms. The automatic application of our method can generate a very large number of potential classifying terms. Any of them can be used to enforce diverse solutions but a manual exploration of the generated classifying terms quickly reveals some that seem more promising than others (in terms of the degree of diversity they could generate).

The definition of a set of heuristics able to filter the set of classifying terms and propose the *best* ones is left for further work.

Combination strategies for classifying terms. Given two or more classifying terms, we could adapt our approach to change the way in which the equivalent classes are traversed in the solution generation process. For instance, we could focus first on one of the classifying terms and generate diverse examples only considering that classifying term alone (i.e. emphasizing *local diversity*). Or we could combine all classifying terms and generate solutions that explore equivalent classes taking into account the value of different classifying terms at the same time (i.e. emphasizing *global diversity*).

6 Conclusions

We have presented a new approach to enforce the generation of a diverse set of instantiations from a given schema. Diversity plays an important role in a variety of scenarios such as model exploration, simulation, testing, validation and verification. In our approach, diversity is guaranteed by the systematic generation of classifying terms that partition the solution space of a model into a set of equivalent classes. Such classifying terms are derived from the strengthening of existing integrity constraints in the schema.

In principle, all constraints can be used as "seed" constraints to generate the CTs. Nevertheless, depending on the application scenario, some constraints are potentially more useful than others. For instance, in a model-based testing

[5] Most solvers will generate by default very similar results when repetitively prompted for new solutions [5, 7, 8]. Rather than (potentially unsuccessful) solver-specific tunings, this work proposes a solver-independent solution to achieve diverse results.

context, one may want to prioritize constraints over the more restricted parts of the model to maximize the chances of finding errors. Identification of such restricted parts/constraints left as future work. As stated in the previous section, we also plan to work on the definition of strategies to optimally select and combine different CTs and guide the exploration of model solutions for each combination. Again, depending on the goal, a breadth-first strategy may be preferable over a depth-first one (or the other way round). Finally, large case studies must check the usefulness of our proposal and improve its applicability.

References

1. Aichernig, B.K., Salas, P.A.P.: Test case generation by OCL mutation and constraint solving. In: QSIC'05. pp. 64–71 (2005)
2. Ali, S., Zohaib Iqbal, M., Arcuri, A., Briand, L.C.: Generating Test Data from OCL Constraints with Search Techniques. IEEE TSE **39**(10), 1376–1402 (2013)
3. Brucker, A.D., Krieger, M.P., Longuet, D., Wolff, B.: A specification-based test case generation method for UML/OCL. In: MODELS'10. pp. 334–348 (2010)
4. Burgueño, L., Clarisó, R., Cabot, J., Gogolla, M.: Constraint mutation source code and examples. `http://hdl.handle.net/20.500.12004/1/C/ER/2019/562` (2019)
5. Chakraborty, S., Fremont, D.J., Meel, K.S., Seshia, S.A., Vardi, M.Y.: On parallel scalable uniform SAT witness generation. In: TACAS'15. pp. 304–319 (2015)
6. Clarisó, R., Cabot, J.: Fixing defects in integrity constraints via constraint mutation. In: QUATIC'18. pp. 74–82 (2018)
7. Dutra, R., Laeufer, K., Bachrach, J., Sen, K.: Efficient sampling of SAT solutions for testing. In: ICSE'18. pp. 549–559 (2018)
8. Ermon, S., Gomes, C., Selman, B.: Uniform solution sampling using a constraint solver as an oracle. In: UAI'12. pp. 255–264 (2012)
9. Ferdjoukh, A., Galinier, F., Bourreau, E., Chateau, A., Nebut, C.: Measurement and generation of diversity and meaningfulness in model driven engineering. International Journal On Advances in Software **11**(1/2), 131–146 (2018)
10. Fleurey, F., Baudry, B., Muller, P.A., Le Traon, Y.: Qualifying input test data for model transformations. SoSyM **8**(2), 185–203 (2007)
11. Gogolla, M., Büttner, F., Richters, M.: USE: A UML-based specification environment for validating UML and OCL. Sci. of Comp. Prog. **69**(1-3), 27–34 (2007)
12. González, C.A., Cabot, J.: Formal verification of static software models in MDE: A systematic review. Information & Software Technology **56**(8), 821–838 (2014)
13. Object Management Group: Object Constraint Language specification (version 2.4), `https://www.omg.org/spec/OCL/2.4/`
14. Hilken, F., Gogolla, M., Burgueño, L., Vallecillo, A.: Testing models and model transformations using classifying terms. SoSyM **17**(3), 885–912 (2018)
15. Jackson, E.K., Simko, G., Sztipanovits, J.: Diversely enumerating system-level architectures. In: EMSOFT'13. pp. 1–10 (9 2013)
16. Olivé, A.: Conceptual Modeling of Information Systems. Springer (2007)
17. Semeráth, O., Varró, D.: Iterative Generation of Diverse Models for Testing Specifications of DSL Tools. In: FASE'18. pp. 227–245 (4 2018)
18. Torlak, E., Jackson, D.: Kodkod: A Relational Model Finder. In: TACAS'07. pp. 632–647 (2007)