JOT
JOURNAL OF
OBJECT TECHNOLOGY

Systematic Engineering of Mutation Operators

Pablo Gómez-Abajo*, Esther Guerra*, Juan de Lara*, and Mercedes G. Merayo†
*Universidad Autónoma de Madrid, Spain
†Universidad Complutense de Madrid, Spain

ABSTRACT In the context of software engineering, *mutation* consists in injecting small changes in artefacts – like models, programs, or data – for purposes like (mutation) testing, test data generation, and all sorts of search-based methods. These tasks typically require defining sets of mutation operators, which are often built ad-hoc because there is currently poor support for their development and testing. To improve this situation, we propose a methodology and corresponding tool support for the proper engineering of mutation operators. Our proposal is model-based, representing the artefacts to be mutated as models. It includes a domain-specific language to describe the mutation operators, facilities to synthesize models that can be used to test the operators, different metrics to analyse operator coverage, and services to generate operators when the coverage is insufficient. We show automated support atop the WODEL tool, and illustrate its use by defining mutation operators for UML Class Diagrams.

KEYWORDS Model-driven engineering; model mutation; model synthesis; metrics; class diagrams; WODEL.

1. Introduction

Mutation consists in the selective introduction of changes into collections of artefacts of a certain type, like models, programs or data. Mutation is at the core of many activities in software engineering, like mutation testing (where programs are injected faults with the purpose of evaluating the quality of a test suite) (DeMillo et al. 1978; Hamlet 1977), test data generation (like in mutation-based fuzzing, which consists in the creation of new input data by introducing small changes to existing test inputs) (Zeller et al. 2019), and search-based software engineering (which applies metaheuristic search techniques to software engineering problems, where candidate solutions are combined and mutated) (Harman & Jones 2001).

Mutation-based methods require the creation of mutation operators able to change the target artefacts in appropriate ways. For example, for mutation testing, operators need to emulate common faults made by competent developers (Papadakis et al. 2019). Such operators are typically defined over the abstract syntax tree of the program, which makes them difficult to test since the input data of the operators are programs. Moreover, mutation operators are often defined ad-hoc using general-purpose programming languages – like Java (Just 2014) or C (Jia & Harman 2008) – not designed for mutating artefacts, which is costly and error-prone. Finally, these sets of mutation operators are usually coded by hand and designed without using a systematic methodology, thus it becomes costly to verify their suitability. For example in mutation testing, effective sets of mutation operators may need to mutate the most important primitives of the language, or the most error-prone (Guerra et al. 2019). Hence, without proper support, it is complicated to assess whether the operator set covers the language as required.

To improve this situation, we propose an integral methodology and supporting tool for the proper engineering of mutation operators. The methodology is model-based to enable its application to heterogeneous artefacts (programs, models, data). This means that the artefacts to mutate are represented as models conforming to a meta-model, for which we rely on injection (artefact-to-model) and extraction (model-to-artefact) transformations. Our solution includes a domain-specific language (DSL) called WODEL (Gómez-Abajo et al. 2016, 2017) specially tailored to design mutation operators applicable over models. To help in the validation of the designed operators, we offer facilities – based on model finding (Jackson 2019) – for synthe-

JOT reference format:
Pablo Gómez-Abajo, Esther Guerra, Juan de Lara, and Mercedes G. Merayo. *Systematic Engineering of Mutation Operators*. Journal of Object Technology. Vol. 19, No. 3, 2020. http://dx.doi.org/10.5381/jot.2020.19.3.a5

sizing models over which the operators can be tested. To assess the coverage of the operators, we provide static and dynamic metrics evaluating the operator *footprints* over the meta-model (static) and over the artefacts where they are applied (dynamic). If the coverage is deemed insufficient, new operators that improve the metrics can be synthesized automatically.

In previous works, we introduced the DSL WODEL (Gómez-Abajo et al. 2016, 2017, 2018a) as well as its basic capabilities for model synthesis (Gómez-Abajo et al. 2020). This paper extends (Gómez-Abajo et al. 2020) by defining a methodology for building operators, facilities for operator synthesis, static and dynamic metrics, and a case study dealing with the mutation of class diagrams.

The rest of this paper is organized as follows. First, Section 2 introduces a running example and overviews our methodology. Then, the following sections illustrate the different ingredients of the methodology. Section 3 describes the WODEL DSL. Section 4 explains our method to synthesize test models for mutation operators, and Section 5 describes our mutation operator footprint metrics. Next, Section 6 shows tool support. Finally, Section 7 analyses related work, and Section 8 presents the conclusions and identifies future lines of work.

2. Motivation and Overview

This section introduces a running example that serves as motivation (Section 2.1), and then provides an overview of our methodology (Section 2.2).

2.1. Running example

Various research works have applied mutation to UML models with the goal to perform mutation testing at the model level (Aichernig et al. 2014; Granda et al. 2016; Krenn et al. 2015; Strug 2016) or to generate input models to MDE programs (i.e., programs created using an engineering approach based on models), as in (de Lara et al. 2019). Following these ideas, as a running example, our goal is to define a set of mutation operators for UML Class Diagrams (UML 2.5.1. Specification by the OMG 2017).

Figure 1 shows part of a simplified UML Class Diagram meta-model for our running example. A class diagram defines a set of Classes, a set of Relations between pairs of classes, a set of Constraints and a set of PrimitiveElements. Classes define a set of Features, which can be Operations or Attributes. There are several types of Relations: Dependencies, Aggregations, Compositions and Associations. Constraints are applied on a Class and expressed in OCL. The figure omits the definition of OCL for clarity. In the meta-model, invariant noInhLoop ensures the absence of inheritance cycles. To illustrate model synthesis in Section 4, we have restricted diagrams to have between 1 and 10 classes (see cardinality of reference ClassDiagram.classes).

Figure 2 shows a UML Class Diagram model in concrete (left) and abstract syntax (right, partially shown). The diagram describes a simple hierarchy of vehicles, which can be either cars or trucks, have a brand and a price, and are made of wheels.

One of the pre-requisites of mutation-based activities is the design of a good set of mutation operators for the given task (e.g., reflecting typical defects of the artefacts under test for mutation testing, or covering the modelling language for test data generation). In this example, we tackle the design of the set of operators proposed in (Granda et al. 2016). This set – made of 50 operators – aims at covering the UML Class Diagram language. However, without proper support, the coverability of the language is challenging to assess because the meta-model is large and one needs to produce many operators. Moreover, once the operators have been designed, there is the need to test whether a particular implementation of them is correct. This task is difficult as it requires building class diagrams for testing each of the 50 operators. To alleviate these issues, the next subsection introduces a tool-supported methodology for the proper engineering of model-based mutation operators.

2.2. Overview of our methodology

Figure 3 shows our process to define mutation operators for a given language. The methodology is independent of the kind of artefact to be mutated. For this purpose, we work with representations of the artefacts as models conformant to a suitable meta-model, and rely on transformations from artefacts to models and vice versa (step 0 in the figure). Creating this meta-model and transformations is not needed if they already exist, as in our running example.

In step 1, the engineer defines the mutation operators. While any programming language can be used to define the operators, we rely on a DSL called WODEL as it provides facilities for defining and performing mutations, as well as support for traceability and analysis. We describe this DSL in Section 3.

In step 2, the engineer validates the defined operators using testing. The goal is to detect possible errors in the operators' implementation by executing them over test models, and checking whether the result is correct. Since creating test models by hand is costly, we provide a facility to synthesize test cases, i.e., models over which the operators are applicable (this seed model synthesis is fully explained in Section 4). This model synthesis is automated by using model finding techniques, and in addition, the engineer can also add manually created test cases.

In step 3, the engineer assesses whether more mutation operators are needed. For this purpose, we offer a suite of dynamic and static footprint metrics that provide a detailed indication of the degree of coverage of the meta-model by the operators. If the coverage is considered insufficient, then, in step 4, the engineer can use our operator synthesizer to automatically generate new operators that increase the coverage. These operators may need to be manually refined, and then tested.

In the following sections, we explain each ingredient of the methodology.

3. The WODEL DSL

WODEL (Gómez-Abajo et al. 2016, 2017) is a DSL targeted to specify mutation operators. The execution of a WODEL program applies the specified operators to a given set of seed models to yield a set of mutant models. For traceability, the execution also produces a registry of the mutations used to produce each mutant. WODEL is domain-independent, and so it can be applied

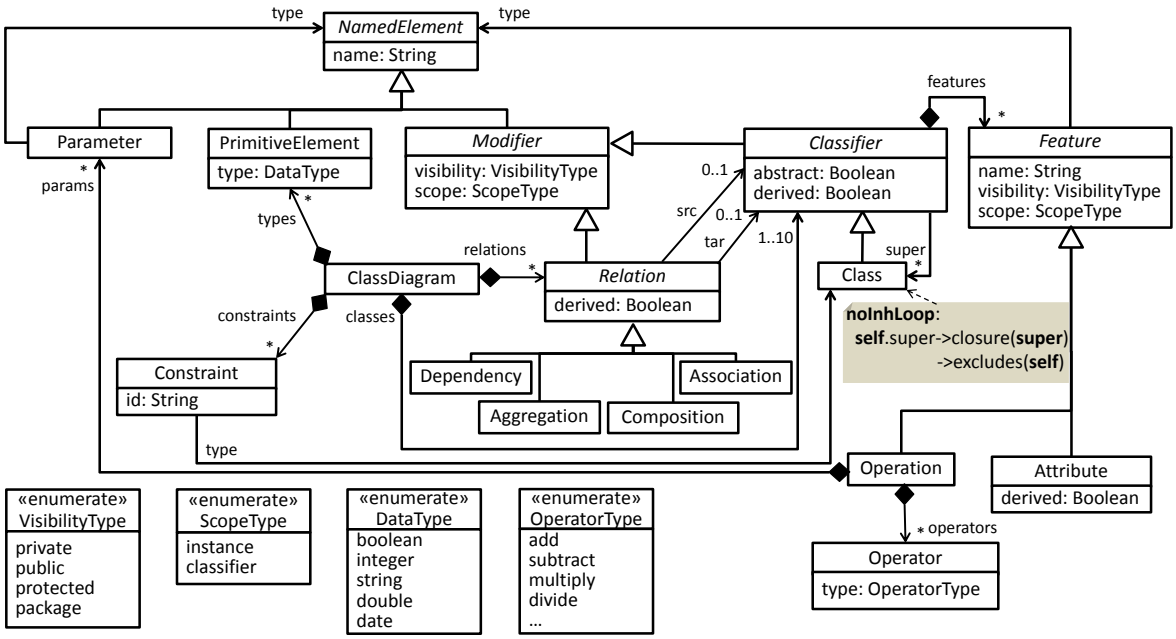

Figure 1 Excerpt of the simplified UML Class Diagram meta-model.

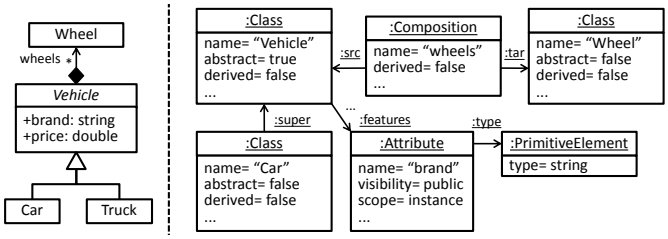

Figure 2 An example UML Class Diagram model using the standard concrete syntax on the left, and a part of the abstract syntax on the right.

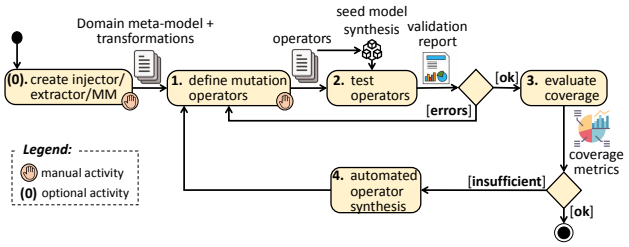

Figure 3 Process for the engineering of mutation operators.

to arbitrary languages, to any kind of artefact which is injected to a model, or to other kinds of artefacts like data. For this purpose, it relies on the provision of a domain meta-model specifying the structure of the artefacts to be mutated. WODEL ensures that the created mutant models conform to the domain meta-model and satisfy its invariants.

WODEL provides mutation primitives to *select*, *modify*, *create*, *delete*, *clone* and *retype* objects; and to *create*, *modify* and *delete* references. In addition, its mutation engine has built-in functionalities to ease the definition of mutation operators. For example, new objects are automatically added to a suitable container reference, and mandatory attributes and references without an explicit value are automatically initialized.

Table 1 illustrates the primitives of WODEL by means of their application to the meta-model of Figure 1. The select operator (row 1) selects a random object with some given feature values. The create operator (row 2) creates an object and initializes its features. If no container reference is specified for the new object, as in the example, then a suitable container is selected at random. The remove operator (row 3) deletes an object and all its incoming and outgoing references. The clone operator (row 4) has two variants: the *shallow* version clones an object but not its contained objects, while the *deep* variant (in the table) clones an object and the elements it contains. In both cases, the object clone is automatically placed in a suitable container. Objects can be retyped (row 5), and the compatible attributes and references are preserved after the retyping. The retype mutation primitive can be applied to sibling objects, or to objects that share any inheritance relation. The objects to be deleted, cloned and retyped can be chosen at random (as in the examples of the table) or according to some matching criteria. In case of retyping, the selection criteria can be based on the object type. As an illustration, the example in row 5 selects an object of one of the types Aggregation, Association, Composition or Dependency, and retypes the object to another of these four types, different from the original type. Object features can be changed using the modify operator (row 6). As for references, they can be created (row 7, which creates a reference of type tar from any Relation object to any Class object), modified (row 8) or deleted (row 9).

Listing 1 shows a simple WODEL program for the mutation of UML Class Diagrams. Line 1 declares the strategy for mutant synthesis, which in this case is exhaustive (i.e., generating all

#	Mutation operator	Example
1	Object selection	c = **select one** Class
2	Object creation	**create** Constraint **with** {type = **one** Class, id = **random**−**string**(1, 4)}
3	Object deletion	**remove one** Association
4	Object cloning	**deep clone one** Class
5	Object retyping	**retype one** [Aggregation, Association, Composition, Dependency] **as** [Aggregation, Association, Composition, Dependency]
6	Feature modification	**modify one** Association **with** {**swap**(src, tar)}
7	Reference creation	**create reference** tar **to one** Class **in one** Relation
8	Reference modification	**modify target** tar **from one** Relation **to other** Class
9	Reference deletion	a = **select one** Association **where** {tar <> **null**} **remove** a→tar

Table 1 Mutation operators provided by WODEL.

possible mutants by the application of the mutation operators as often as possible). Instead, it is possible to specify a maximum number of mutants. Line 2 specifies the output folder to store the mutants, and the input folder that contains the seed models. Our implementation is based on EMF (i.e., the Eclipse Modeling Framework), and hence, line 3 indicates the URI or location of the Ecore meta-model in use. The remainder of the program defines the mutation operators. In this example, the operator rel2rel retypes an Aggregation, Association or Composition into another of these three types (lines 6–9). This operator only uses a mutation primitive, but in general, operators can use any number of primitives and instructions. Mutation primitives can be scheduled to be applied a random number of times within a given interval; if they do not define an interval (as in the example), they are applied once. Finally, a WODEL program may include OCL invariants that any generated mutant must satisfy. The appendix contains the encoding of other more complex operators for UML Class Diagrams proposed in (Granda et al. 2016).

```
1  generate exhaustive mutants
2  in "out/" from "model/"
3  metamodel "http://umlcd.com"
4
5  with blocks {
6    rel2rel {
7      retype one [Aggregation, Association, Composition]
8      as [Aggregation, Association, Composition]
9    }
10 }
```

Listing 1 WODEL program for the UML Class Diagram meta-model in Figure 1.

Figure 4 shows an application of the mutation operator in Listing 1 to a UML Class Diagram. In the resulting mutant, the Composition 'wheels' becomes replaced by an equally named Association.

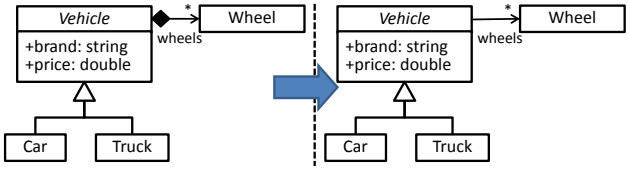

Figure 4 Application of mutation operator to a UML Class Diagram model.

4. Seed Model Synthesis

To ease the testing of the operators defined in WODEL programs, we enable an automated synthesis of seed models over which all instructions of the given program are applicable (if such models exist in the given search scope).

Figure 5 outlines the seed model synthesis process. It relies on model search, a technique which applies constraint resolution over models (Jackson 2019). In particular, the synthesizer enriches the description of the domain meta-model and its invariants with additional OCL constraints derived from the WODEL program. These constraints express the requirements that a seed model must fulfil to allow the application of each mutation operator included in the program. Next, the enriched meta-model is loaded into a model finder (like the USE Validator (Kuhlmann et al. 2011; Kuhlmann & Gogolla 2012)) which performs a bounded search of instances of the meta-model satisfying the OCL constraints. If a model is found, then it ensures full statement coverage of the WODEL program when executed with the model.

Table 2 shows the templates used to generate the OCL constraints for each mutation primitive, and illustrative examples. For instance, the OCL template for the object deletion primitive demands the existence of an object with the specified type and feature values, and included in a container reference that would not violate its lower cardinality bound if the object deletion

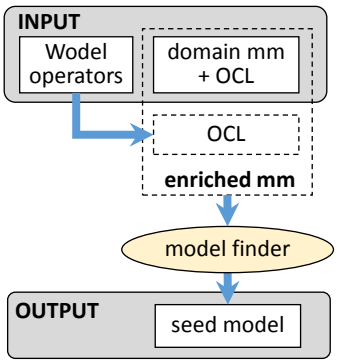

Figure 5 Automated model synthesis for testing mutation operators.

takes place. The table shows as an example the deletion of a Class: the derived OCL constraint checks that there exists some Class, and the ClassDiagram to which it belongs contains other classes in addition to this one (i.e., the size of reference ClassDiagram.classes is bigger than 1, and deleting the Class would still satisfy the reference cardinality). Other OCL templates deal with object creation (which requires the existence of a suitable container reference with enough space for the object), object cloning (which in addition requires the existence of a candidate object to be cloned), object retyping (which requires conditions equivalent to those for deleting and creating objects for every container or regular reference that is not source- or target-compatible with the new type), reference modification (which requires the existence of an object of the target class), reference creation (which in addition requires a reference with space to add the object of the target class), and reference deletion (which requires that the reference fulfils its lower cardinality after taking one of its objects out).

For readability, Table 2 shows the template associated to one occurrence of a mutation primitive. However, a program may apply the same primitive with the same parameters more than once. This may occur because the primitive is repeated, or because it defines an interval of applications bigger than one. Hence, in the general case, we count how many times a same instruction appears (i.e., is to be executed), and generate a slightly more complex constraint where each such occurrence is represented as a variable. For instance, if the mutation create Class appears twice in a program, we generate the following constraint (cf. Table 2):

```
ClassDiagram.allInstances()→exists(c1,c2 |
  (c1 <> c2 and c1.classes→size() < 10
   and c2.classes→size() < 10)
  or c1.classes→size() < 9)
```

Overall, the model synthesis process starts with the domain meta-model and its invariants. The meta-model is added an auxiliary mandatory class named Dummy. Then, the process uses the templates of Table 2 to generate the OCL constraints for each mutation operator in the provided WODEL program. These constraints are added as invariants of the Dummy class. Finally, the model finder is invoked with this enriched meta-model as input.

As an example, Listing 2 shows the OCL constraint generated from the program in Listing 1. As the retype operation considers the retyping of objects of three different types, an or with three cases is generated. Figure 6 shows a seed model returned by the model finder. This conforms to the original meta-model and satisfies the constraint of Listing 2.

```
1 context Dummy
2
3 inv mut1 :
4   Aggregation.allInstances()→exists(a | true) or
5   Association.allInstances()→exists(a | true) or
6   Composition.allInstances()→exists(a | true)
```

Listing 2 Constraint derived from Listing 1.

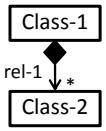

Figure 6 Generated seed model.

Note that the synthesized seed models enable the application of all statements in the WODEL program. However, they do not guarantee that, after applying the program, the resulting mutant satisfies the invariants of the domain meta-model. This would require from techniques for advancing constraints to model operations (Sánchez Cuadrado et al. 2017), which is left for future work.

5. Mutation Operator Footprints

In order to assess the coverage of the designed mutation operator set, we support two kinds of footprints: static (cf. Section 5.1) and dynamic (cf. Section 5.2).

5.1. Static footprints

Static footprints collect which meta-model elements can become affected by a WODEL program, and which kind of actions (creation, modification, or deletion) can be performed on them. This is useful to statically analyse whether a set of mutation operations is complete with respect to the meta-model, or at least covers the mutation of all meta-model elements of interest for a given application domain. This help is very valuable when the meta-model is large or there are many mutation operations to consider.

The static footprint information can be aggregated according to two criteria. On the one hand, *meta-model static footprints* aggregate the information for each meta-model element (class or feature). On the other, *operator static footprints* aggregate the information for each mutation operator defined in a program. In this way, the footprint reports on the number of times each kind of element is created, modified or deleted by some code sentence, and percentages with respect to the total number of elements are calculated as well.

Regarding the collected information, we distinguish between *explicit* and *implicit* actions. The former are actions explicitly

Conditions to check	OCL template	Example	
Object filter: Auxiliary template used to check that an object has the given feature values.	o.⟨feat$_1$⟩ = ⟨val$_1$⟩ ... **and** o.⟨feat$_n$⟩ = ⟨val$_n$⟩		
Object selection, **object modification**: There is an object with the given type and feature values.	⟨Class⟩.**allInstances**() →**exists**(o | ⟨object−filter⟩)	**Wodel**: **modify one** Class **where** {name = 'InitialName'} **with** {name = 'ModifiedName'}	**OCL**: Class.**allInstances**() →**exists**(c | c.name = 'InitialName')
Object creation: There is a container reference of the object's type with space to add more objects.	⟨Container⟩.**allInstances**() →**exists**(o | o.⟨ref⟩→**size**() < ⟨upB⟩)	**Wodel**: c = **create** Class	**OCL**: ClassDiagram.**allInstances**() →**exists**(d | d.classes→**size**() < 10)
Object deletion: There is an object with the given type and feature values, and its deletion does not violate the lower bound of any reference of the object's type.	⟨Class⟩.**allInstances**() →**exists**(o | ⟨object−filter⟩ **and** ⟨Container⟩.**allInstances**() →**forAll**(c | c.⟨ref⟩→**includes**(o) **implies** c.⟨ref⟩→**size**() > ⟨lowB⟩))	**Wodel**: **remove one** Class	**OCL**: Class.**allInstances**()→**exists**(c | ClassDiagram.**allInstances**() →**forAll**(d | d.classes→**includes**(c) **implies** d.classes→**size**() > 1))
Object cloning: There is an object with the given type and feature values, and a container reference of that type with space to add more objects.	⟨Class⟩.**allInstances**() →**exists**(o | ⟨object−filter⟩) **and** ⟨Container⟩.**allInstances**() →**exists**(o | o.⟨ref⟩→**size**() < ⟨upB⟩)	**Wodel**: **deep clone one** Class	**OCL**: Class.**allInstances**() →**exists**(c | true) **and** ClassDiagram.**allInstances**() →**exists**(d | d.classes→**size**() < 10)
Object retyping: There is an object with the given source type and feature values. If the target type is not compatible with the container of the source type, conditions to delete a source object and create a target one are required (and similar for refs not compatible with target type). Or-catenate for each considered source/target type.	⟨Class⟩.**allInstances**() →**exists**(o | ⟨object−filter⟩ [**and** ⟨SrcContainer⟩.**allInstances**() →**forAll**(c | c.⟨ref⟩→**includes**(o) **implies** c.⟨ref⟩→**size**() > ⟨lowB⟩) **and** ⟨TrgContainer⟩.**allInstances**() →**exists**(c | c.⟨ref⟩→**size**() < ⟨upB⟩)]^1) ^{1}add condition if ⟨SrcContainer⟩.⟨ref⟩ is not compatible with target type	**Wodel**: **retype one** Composition **as** Association	**OCL**: Composition.**allInstances**() →**exists**(c | true)
Reference creation: There is an object of the reference type, and a reference to which we can add the object without violating the upper bound.	⟨TgtClass⟩.**allInstances**() →**exists**(o | ⟨object−filter⟩) **and** ⟨SrcClass⟩.**allInstances**() →**exists**(o | ⟨object−filter⟩ **and** o.⟨ref⟩→**size**() < ⟨upB⟩)	**Wodel**: **create reference** src **to one** Class **in one** Composition	**OCL**: Class.**allInstances**() →**exists**(c | true) **and** Composition.**allInstances**() →**exists**(c | c.src→**size**() < 1)
Reference modification: There is a non-empty reference of the given kind, and more than one object of the reference target type.	⟨SrcClass⟩.**allInstances**() →**exists**(o | o.⟨ref⟩→**notEmpty**()) **and** ⟨TgtClass⟩.**allInstances**() →**size**() > 1	**Wodel**: **modify target** tar **from one** Composition **to other** Class	**OCL**: Composition.**allInstances**() →**exists**(c | c.tar→**notEmpty**()) **and** Class.**allInstances**() →**size**() > 1
Reference deletion: There is a reference from which we can remove an object without violating the lower bound.	⟨Class⟩.**allInstances**() →**exists**(o | ⟨object−filter⟩ **and** o.⟨ref⟩→**size**() > ⟨lowB⟩)	**Wodel**: c = **select one** Composition **remove** c→tar	**OCL**: Composition.**allInstances**() →**exists**(c | c.tar→**size**() > 0)

Table 2 Templates to generate OCL constraints from mutation primitives.

Explicit action	Derived implicit actions
Creation of object of class C	– creation of superclasses of C – creation of mandatory attributes defined by C or a supertype – creation of mandatory references defined by C or a supertype – modification of containers of C or a supertype
Modification of object of class C	*none*
Deletion of object of class C	– deletion of subclasses of C – deletion of attributes defined by C or a supertype – deletion of references defined by C or a supertype – modification of references that point to C or a supertype
Cloning of object of class C	*Shallow cloning*: – creation of superclasses of C – creation of mandatory attributes defined by C or a supertype – creation of mandatory references defined by C or a supertype – modification of containers of C or a supertype *Deep cloning*: – shallow cloning of C – for each class C' reachable from C, shallow cloning of C'
Retyping of object of class C	– implicit actions for creation of object of class C – implicit actions for deletion of object of class C
Creation of feature	*none*
Modification of feature	*none*
Deletion of feature	*none*

Table 3 Rules to calculate implicit actions.

defined in a WODEL program, encoded as mutation primitives acting on the instances of some meta-model element. For example, the instruction remove one Class from assoc→src explicitly removes an object from reference Association.src, and hence, the static footprint would count an explicit modification of this reference. If there are other explicit modifications of the reference in the same program, this counter is increased accordingly. In contrast, implicit actions are side effects on the models due to explicit actions. For example, mutation remove one Composition removes an object of type Composition explicitly, but in addition, it implicitly removes its references Compositon.src and Composition.tar, and modifies the container reference ClassDiagram.relations by taking the removed composition object out. This way, this information uncovers hidden actions of mutation operators.

Table 3 lists the implicit actions that may occur as a result of an explicit action. Only the creation, deletion, cloning and retyping of objects may entail implicit actions. The computation of the implicit actions takes into account the inheritance relations in the domain meta-model. For instance, when an operator specifies the creation of an object of type C, we increase the implicit creation count for all superclasses of C because objects of type C are compatible with the ancestors of C. Likewise, when an operator deletes objects of type C, we increase the implicit deletion count for all subclasses of C, as the operator could delete objects of the subclasses of C as well. As explained in Section 3, WODEL automatically initializes the mandatory features of created objects for which no explicit value is specified. For this reason, creating an object implies the implicit creation of its owned and inherited mandatory features, if they lack an explicit value. The retyping of objects is considered as an object deletion followed by an object creation, so it entails the implicit actions of both. Finally, if a WODEL program specifies an action explicitly, then it is counted as explicit but not as implicit.

Table 4 shows the meta-model static footprint of the set of UML Class Diagram mutation operators in the Appendix. This static footprint shows the meta-model coverage by this set of mutation operators. The table columns show the explicit and implicit creation, modification and deletion actions over each meta-model class and feature. The rows corresponding to classes show the number of actions performed on the class, as well as the aggregated actions on its features. For instance, there are two explicit creations of class Parameter, and one explicit creation of its features; therefore, the cell for the explicit creation of class Parameter contains 2c 1f. The third row of the table (Class coverage in bold) shows the average class coverage for each type of action. For example, the explicit creation percentage is 22% because the program explicitly creates 11 out of the 49 existing meta-model concrete classes.

Overall, these metrics provide information about possible

	Explicit			Implicit		
Classes **Features**	**C**	**M**	**D**	**C**	**M**	**D**
Class coverage	**22%**	**16%**	**26%**	**26%**	**22%**	**63%**
▷ ClassDiagram					42c 42f	
▷ PrimitiveElement						
▼ Parameter	2c 1f	1c 1f	1c 0f	1c 5f		2c 6f
(a) name	1			2		3
(r) type		1		3		3
▷ Class	3c 1f	6c 6f	2c 0f	0c 14f		0c 16f
▷ Attribute	2c 2f	5c 4f	1c 0f	0c 8f		0c 5f
▷ Constraint	2c 2f		1c 0f	0c 2f	2c 2f	0c 3f
▷ Operation	2c 2f	4c 4f	2c 0f	0c 6f	9c 9f	0c 12f
▷ Operator				1c 1f		2c 2f
▷ Dependency	3c 0f		3c 0f	0c 18f		0c 24f
▷ Association	7c 13f	2c 3f	4c 0f	0c 31f		0c 32f
▷ Aggregation	3c 0f		3c 0f	0c 18f		0c 24f
▷ Composition	3c 0f		3c 0f	0c 18f		0c 24f

Table 4 Meta-model static footprint excerpt of the UML Class Diagram mutation operators set. C=Creation, M=Modification, D=Deletion, (a)=attribute, (r)=reference.

language coverage gaps in the operator set. In our example, we can realize that there are no operators explicitly targeting classes ClassDiagram, PrimitiveElement or Operator.

5.2. Dynamic footprints

Static footprinting provides an over-approximation of the possible effects of a mutation program. It is an over-approximation because some effects will depend on the particular seed model the program is applied to. For example, when a mutation deletes an object, the static footprint reports that all reference types that might include the object are implicitly modified; however, in a particular model, the object may be included only in some of them.

Hence, we introduce dynamic footprints to analyse the actual effects that a mutation program has on specific seed models. We distinguish two kinds of dynamic footprints. On the one hand, *net dynamic footprints* summarize the net effect of a mutation program by differencing the seed model and the resulting mutant. On the other hand, *debug dynamic footprints* are more detailed, as they record the specific mutation operators applied by the program. This is useful to detect situations where a mutation operator cancels the effects of previous operators (e.g., an object is created by an operator and then deleted by a subsequent operator). In such cases, debug footprints will reflect the mutations being cancelled, while net footprints do not provide such a level of detail. Thus, we can consider that the net dynamic footprints reflect a state-based comparison of models, and the debug dynamic footprints reflect an operation-based comparison, as explained in detail in (Brosch et al. 2012).

6. Tool Support

The development environment for WODEL is available as an Eclipse plugin at http://miso.es/tools/Wodel.html, together with examples and videos. The implementation is based on EMF (Steinberg et al. 2008), and expects the meta-models of the artefacts to be mutated to be specified using Ecore. The environment (Gómez-Abajo et al. 2018a) features code completion and type checking, and the semantics of WODEL programs is given by their compilation into Java code. The environment can be extended with post-processing applications. We have developed two such extensions for the automated generation of exercises (Gómez-Abajo et al. 2017) and the creation of mutation testing tools (Gómez-Abajo et al. 2019).

To support the contributions presented in this paper, we have extended the WODEL IDE to support the synthesis of seed models for testing mutation programs, the computation and visualization of mutation footprints, and the automated generation of mutation operators that improve the footprints.

The synthesis of seed models for a given program is configured by means of the wizard shown in Figure 7. This allows setting the maximum number of seed models to be generated, the mutation operators used in the seed model generation process (either all operators in the program or a subset), additional model requirements expressed by OCL, and optionally, an EMF model to be used as seed of the model search. Moreover, a preference page allows customizing the minimum and maximum number of objects and references that the produced seed models should have. The search of seed models is performed with the USE Validator (Kuhlmann & Gogolla 2012) model finder.

The IDE provides four views for the footprints introduced in Section 5: meta-model static, operator static, net dynamic, and debug dynamic. Figure 8 shows a screenshot of the tool with these views at the bottom, for the set of mutation operators in the Appendix. The views contain drill-drown tables with the footprint information, and use different cell colours to distinguish between creation, modification and deletion of classes and features. The table headers show a percentage that aggregates the information in the table. For meta-model static footprints, this is the percentage of classes whose instances are created, modified or deleted by the WODEL program, and can be calculated either relative to all meta-model classes, or only to concrete classes in order to better convey the meta-model coverage.

Meta-model static footprints can also be visualized on the meta-model (specifically, on top of a Sirius-based (Sirius 2017) visualization of the Ecore meta-model). Figure 9 shows the running example meta-model annotated with the footprint information. We use a traffic light metaphor where classes and attributes include icons with different colours to indicate the explicit creation (green), modification (amber) or deletion (red) of their instances. These colours are the same used in the footprint views (see Figure 8). In the example, classes Class, Operation, Attribute, Constraint, Parameter, Dependency, Aggregation, Composition and Association are explicitly mutated, and hence, only these classes are annotated with icons. Likewise, explicitly mutated references are shown with a different colour. This is the

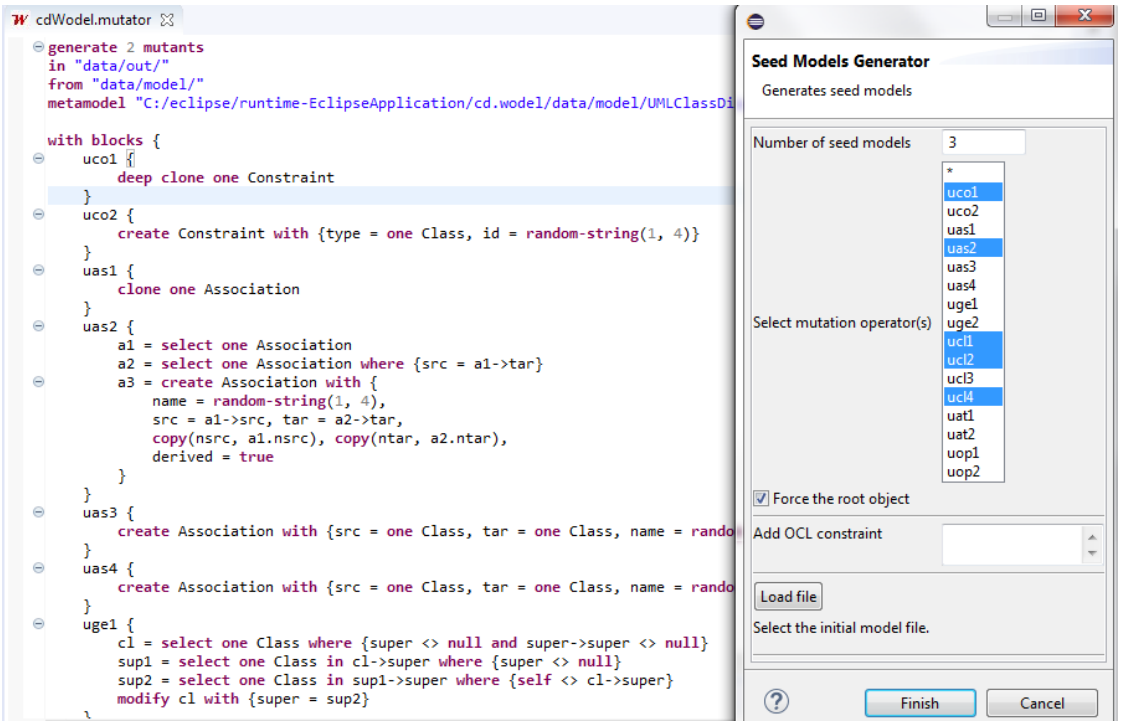

Figure 7 Seed model synthesizer.

case for references Constraint.type and Operation.params.

If the footprint evinces a missing that some mutation operator is missing, then its semi-automatic creation is possible. For this purpose, the view for meta-model static footprints integrates a wizard that can be launched by double-clicking on a meta-model class or feature name, or on a cell of the footprint table. The wizard allows selecting one among the mutation primitives that solve the missing coverage, together with its execution details. Then, the wizard automatically extends the original WODEL program with the new mutation operation. Figure 8 shows the wizard on the right-top being used to create a mutation operator for the uncovered meta-model class PrimitiveElement.

7. Related Work

To the best of our knowledge, there is no comprehensive solution for the engineering of mutation operators. Hence, in the following, we review works related to the four main ingredients of our methodology: languages tailored to define mutation operators, operator synthesis, model synthesis from requirements, and approaches to calculate static metrics of transformations. Finally, we also review works applying mutation to UML diagrams.

DSLs for mutation operators. Some model-based mutation approaches use general-purpose model transformation languages to define mutation operators. In (González et al. 2018), the authors present an MDE approach to define mutation testing tools, where programs are represented as models, and operators are encoded in QVT-o. The approach introduced in (Mottu et al. 2006) uses mutation analysis to select an appropriate set of mutation operators for model transformation. The authors establish a criteria based on semantic faults in the navigation, the filtering, the output model creation, and the input model modification to create such set. Mutation operators have also been defined using Henshin in (Burdusel et al. 2019), and ATL in (Troya et al. 2015). Instead, WODEL is a DSL targeted to define mutation operators, giving support for specific mutation actions (e.g., retyping, cloning), the automatic initialization of object features and containers, and the configuration of the number of mutants to generate. Works like (Troya et al. 2015) miss such policies and only produce one mutant per input model.

Major (Just 2014) is a mutation testing tool for Java that includes a scripting language to perform small customizations in mutation operators. For example, it allows configuring the replacement lists of mutation operators like Arithmetic Operator Replacement (AOR). Instead, WODEL is more expressive as it enables the selection, creation, deletion and retyping of elements. Moreover, WODEL is language-independent, as one can define operators for arbitrary meta-models. WODEL can also be used to mutate Java programs as we introduced in our previous work (Gómez-Abajo et al. 2018b).

Mutation operator synthesis. In (Alhwikem et al. 2016), the authors propose a set of mutation primitives to define mutation operators for Ecore meta-models. However, it is not a full-fledged DSL, missing essential features like the possibility of selecting elements, and there is no tool support for execution. The approach in (Burdusel et al. 2019) generates operators that guarantee the consistency of the mutated models with the meta-model multiplicity constraints. The methodology presented in (Kehrer et al. 2016) automatically generates a set of mutation operators according to given patterns based on the meta-model types and relations. In both of these works, the operators are encoded as graph transformation rules. In comparison, WODEL

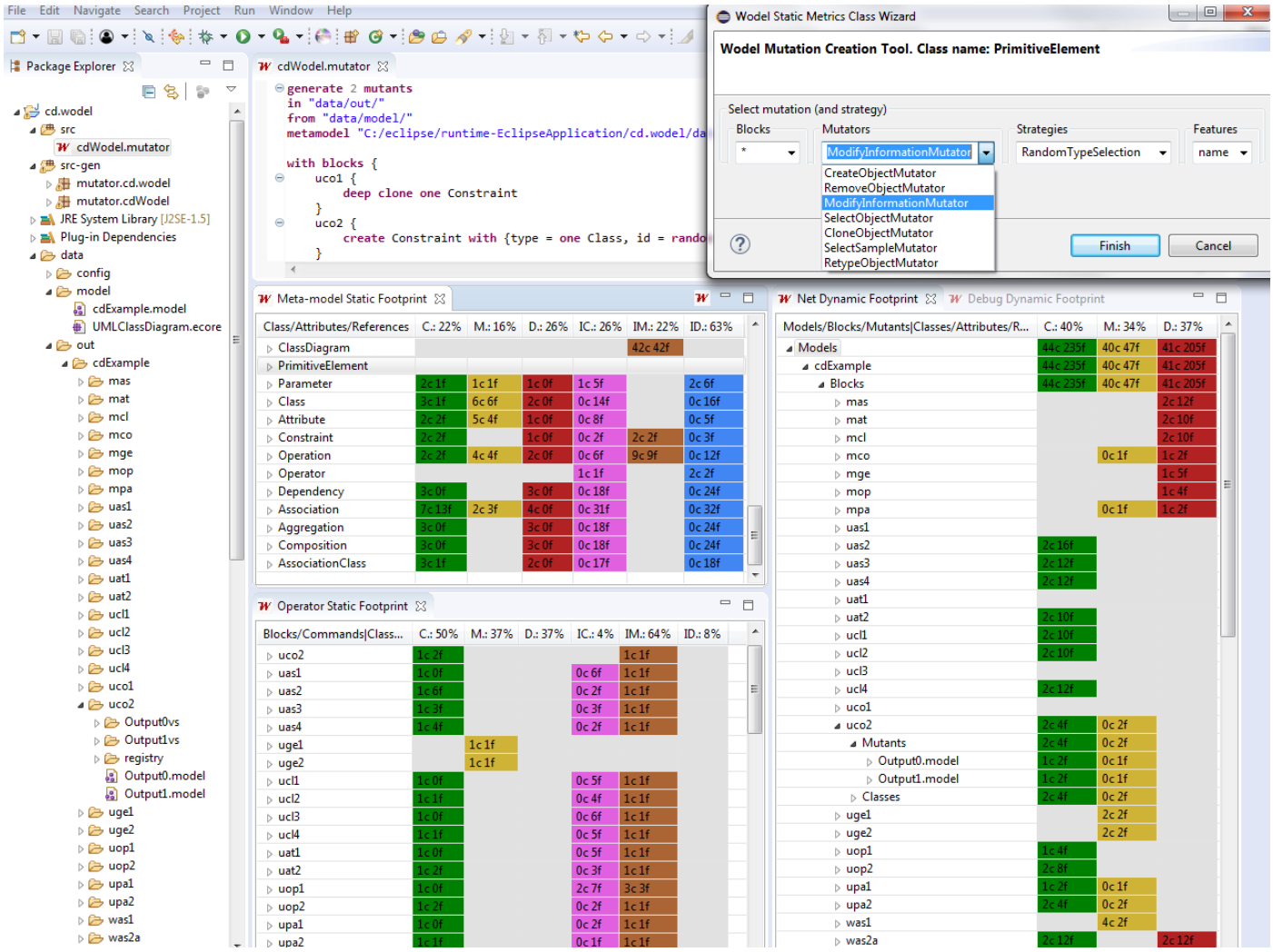

Figure 8 Footprint views and wizard for automated mutation operator creation.

considers more advanced primitives, like cloning, modifying the source or target of references, and retyping. Our techniques for model synthesis (for testing) could be a complement to these two approaches.

Model synthesis. The MDE community has used model finders (like USE (Kuhlmann & Gogolla 2012) or Alloy (Jackson 2019)) for activities like model completion, test model generation, or transformation analysis. For example, model finding is used in (Guerra & Soeken 2015; Hilken et al. 2018) to generate test models for model transformations by translating the transformation specifications into OCL, while (Guerra et al. 2020) uses model finding to efficiently analyse product lines of meta-models. The approach proposed in (Fleurey et al. 2004) selects the test models for a given model transformation from a set of models generated by a bounded search of all the possible combinations of the input meta-model elements. However, this methodology does not assure the program applicability over the generated models. In our case, the novelty yields in the encoding of the semantics of WODEL programs into OCL, ensuring full statement coverage of the program.

Static metrics for transformations. In the model transformation area, static metrics have been used to ease the maintainability of transformations (van Amstel & van den Brand 2011). Model transformations have also been analysed statically to detect rule conflicts (Ehrig et al. 2006), violations of transformation contracts (Burgueño et al. 2015), or typing errors (Sánchez Cuadrado et al. 2017; Ujhelyi et al. 2011). Our static footprints are a kind of static analysis tool to detect meta-model elements not covered by a mutation program, as well as missing mutation operators. Further static analyses, like detecting operator conflicts, are future work.

Mutation for UML. Some approaches have proposed model-level mutation for UML. For example, (Granda et al. 2016) proposes 50 mutation operators for class diagrams, which we have used in our running example. The approach is backed by a tool, called mutUML, which is specific to UML. Instead, WODEL is applicable to arbitrary modelling languages, and provides metrics and facilities for input model synthesis.

MoMuT::UML (Aichernig et al. 2014; Krenn et al. 2015) is a tool to mutate UML models (class, state charts and instance diagrams). It is a black-box test case generation tool that specialises in fault-based test case generation. More precisely, the

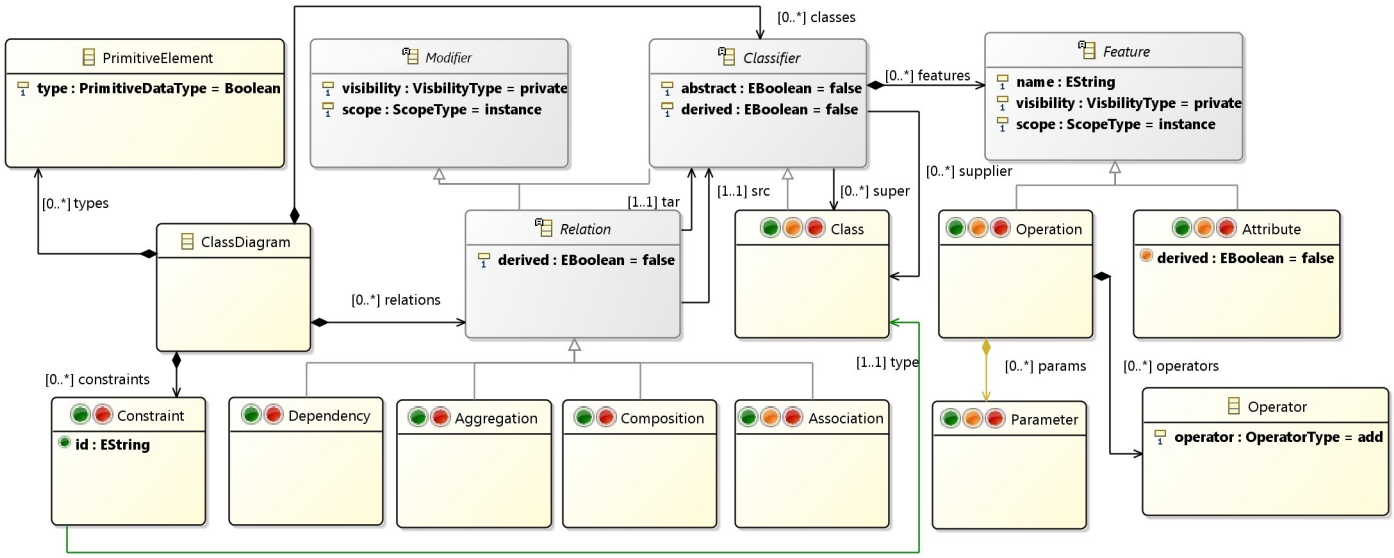

Figure 9 Meta-model with static footprint.

tool generates model mutants and is able to find test cases that reveal them. Again, it is specific to UML, but we believe that an approach like the one we propose here would ease its extension with new operators.

In (Strug 2016), the author compares the reliability of mutation testing at the model and implementation levels. For this purpose, at the model level, the author developed a set of mutation operators for UML/OCL, and used USE command scripts (Gogolla et al. 2007) for testing. At the implementation level, the author used the muJava mutation tool (Ma et al. 2006). Overall, the study revealed that assessing the quality of a test suite at the implementation level can already be made at the model level. In this setting, our approach could be used to facilitate the creation of model mutation operators.

8. Conclusions and Future Work

Given the recurrent need to develop sets of mutation operators, this paper has proposed a methodology and tool support for their proper engineering. Specifically, we provide a DSL for describing mutation operators, model synthesis capabilities for their testing and validation, metrics for the measurement of their completeness, and automated mutation operator generation to augment the operator set. We have illustrated the approach in the context of UML Class Diagrams.

We are currently extending the model synthesis process in two ways. First, to generate models where the operators are not applicable but are close to being applicable, so called near misses (Montaghami & Rayside 2017). Second, to generate seed models ensuring that the execution of the WODEL program yields a mutant model that satisfies all invariants in the domain meta-model. For this purpose, we may use techniques to advance OCL constraints as preconditions, based on (Sánchez Cuadrado et al. 2017). We plan to work on further static analysis techniques, e.g., to detect operator conflicts and dependencies. Finally, we aim at developing techniques for reusing mutation operators for other meta-models different from the one they were designed for, in the style of some of the techniques compared in (Bruel et al. 2020). This would allow, for example, applying the operators of the running example for Ecore meta-models.

Acknowledgments

This work has been partially funded by the Spanish Ministry of Science (RTI2018-095255-B-I00), by the R&D programme of the Madrid Region (S2018/TCS-4314) and by the Spanish MINECO-FEDER (grant number FAME RTI2018-093608-B-C31).

We are grateful to Martin Gogolla and his team for the continued maintenance and improvement of the USE modelling tool (Gogolla et al. 2007, 2020), which has been widely used by the modelling community and our team, and has been a crucial part of the work presented in this paper.

References

Aichernig, B. K., Auer, J., Jöbstl, E., Korosec, R., Krenn, W., Schlick, R., & Schmidt, B. V. (2014). Model-based mutation testing of an industrial measurement device. In *Tap* (Vol. 8570, pp. 1–19). Springer.

Alhwikem, F., Paige, R. F., Rose, L., & Alexander, R. (2016). A systematic approach for designing mutation operators for MDE languages. In *MODEVA* (Vol. 1713, pp. 54–59). CEUR-WS.org.

Brosch, P., Kappel, G., Langer, P., Seidl, M., Wieland, K., & Wimmer, M. (2012). An introduction to model versioning. In *12th international conference on formal methods for the design of computer, communication, and software systems* (p. 336–398). Springer-Verlag.

Bruel, J., Combemale, B., Guerra, E., Jézéquel, J., Kienzle, J., de Lara, J., … Vangheluwe, H. (2020). Comparing and classifying model transformation reuse approaches across metamodels. *Software and Systems Modeling*, *19*(2), 441–465.

Burdusel, A., Zschaler, S., & John, S. (2019). Automatic generation of atomic consistency preserving search operators for search-based model engineering. In *MODELS* (pp. 106–116). IEEE.

Burgueño, L., Troya, J., Wimmer, M., & Vallecillo, A. (2015). Static fault localization in model transformations. *IEEE Trans. Software Eng.*, *41*(5), 490–506.

de Lara, J., Guerra, E., Ruscio, D. D., Rocco, J. D., Sánchez Cuadrado, J., Iovino, L., & Pierantonio, A. (2019). Automated reuse of model transformations through typing requirements models. *ACM Trans. Softw. Eng. Methodol.*, *28*(4), 21:1–21:62.

DeMillo, R. A., Lipton, R. J., & Sayward, F. G. (1978). Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, *11*(4), 34–41.

Ehrig, H., Ehrig, K., Prange, U., & Taentzer, G. (2006). *Fundamentals of algebraic graph transformation*. Springer-Verlag.

Fleurey, F., Steel, J., & Baudry, B. (2004). Validation in model-driven engineering: testing model transformations. In *First international workshop on model, design and validation* (p. 29-40).

Gogolla, M., Büttner, F., & Richters, M. (2007). USE: A UML-based specification environment for validating UML and OCL. *Sci. Comput. Program.*, *69*(1-3), 27–34.

Gogolla, M., Havakili, H., & Schipke, C. (2020). Advanced features for model visualization in the UML and OCL tool USE. In *Companion proceedings of modellierung 2020* (Vol. 2542, pp. 203–207). CEUR-WS.org.

Gómez-Abajo, P., Guerra, E., & de Lara, J. (2016). Wodel: a domain-specific language for model mutation. In *Sac* (pp. 1968–1973). ACM.

Gómez-Abajo, P., Guerra, E., & de Lara, J. (2017). A domain-specific language for model mutation and its application to the automated generation of exercises. *Computer Languages, Systems & Structures*, *49*, 152 - 173.

Gómez-Abajo, P., Guerra, E., de Lara, J., & Merayo, M. (2020). Seed model synthesis for testing model-based mutation operators. In *CAISE FORUM* (pp. 1–12).

Gómez-Abajo, P., Guerra, E., de Lara, J., & Merayo, M. G. (2018a). A tool for domain-independent model mutation. *Sci. Comput. Program.*, *163*, 85–92.

Gómez-Abajo, P., Guerra, E., de Lara, J., & Merayo, M. G. (2018b). Towards a model-driven engineering solution for language independent mutation testing. In *Jisbd* (p. 4pps). Biblioteca digital SISTEDES.

Gómez-Abajo, P., Guerra, E., de Lara, J., & Merayo, M. G. (2019). Mutation testing for DSLs (tool demo). In *Dsm* (pp. 60–62). ACM.

González, A., Luna, C., & Bressan, G. (2018). Mutation testing for Java based on model-driven development (in spanish). In *Clei-slisw.*

Granda, M. F., Condori-Fernández, N., Vos, T. E. J., & Pastor, O. (2016). Mutation operators for UML Class Diagrams. In *Advanced information systems engineering* (pp. 325–341). Springer International Publishing.

Guerra, E., de Lara, J., Chechik, M., & Salay, R. (2020). Property satisfiability analysis for product lines of modelling languages. *IEEE Trans. Software Eng.*, *In press*. doi: http://dx.doi.org/10.1109/TSE.2020.2989506

Guerra, E., Sánchez Cuadrado, J., & de Lara, J. (2019). Towards effective mutation testing for ATL. In *Models* (pp. 78–88). IEEE.

Guerra, E., & Soeken, M. (2015). Specification-driven model transformation testing. *Software and Systems Modeling*, *14*(2), 623–644.

Hamlet, R. G. (1977). Testing programs with the aid of a compiler. *IEEE Trans. Software Eng.*, *3*(4), 279–290.

Harman, M., & Jones, B. F. (2001). Search-based software engineering. *Information & Software Technology*, *43*(14), 833–839.

Hilken, F., Gogolla, M., Burgueño, L., & Vallecillo, A. (2018). Testing models and model transformations using classifying terms. *Software and Systems Modeling*, *17*(3), 885–912.

Jackson, D. (2019). Alloy: a language and tool for exploring software designs. *Commun. ACM*, *62*(9), 66–76.

Jia, Y., & Harman, M. (2008). MILU: A customizable, runtime-optimized higher order mutation testing tool for the full C language. In *Taicpart* (pp. 94–98).

Just, R. (2014). The Major mutation framework: Efficient and scalable mutation analysis for Java. In *Issta* (pp. 433–436). ACM.

Kehrer, T., Taentzer, G., Rindt, M., & Kelter, U. (2016). Automatically deriving the specification of model editing operations from meta-models. In P. V. Gorp & G. Engels (Eds.), *Theory and practice of model transformations* (pp. 173–188). Springer International Publishing.

Krenn, W., Schlick, R., Tiran, S., Aichernig, B. K., Jöbstl, E., & Brandl, H. (2015). Momut: UML model-based mutation testing for UML. In *Icst* (pp. 1–8). IEEE Computer Society.

Kuhlmann, M., & Gogolla, M. (2012). From UML and OCL to relational logic and back. In *MODELS* (Vol. 7590, pp. 415–431). Springer.

Kuhlmann, M., Hamann, L., & Gogolla, M. (2011). Extensive validation of OCL models by integrating SAT solving into USE. In *Objects, models, components, patterns - 49th international conference, TOOLS* (Vol. 6705, pp. 290–306). Springer.

Ma, Y.-S., Offutt, A. J., & Kwon, Y. R. (2006). MuJava: a mutation system for Java. In *Icse* (pp. 827–830).

Montaghami, V., & Rayside, D. (2017). Bordeaux: A tool for thinking outside the box. In *FASE* (Vol. 10202, pp. 22–39). Springer.

Mottu, J.-M., Baudry, B., & Traon, Y. L. (2006). Mutation analysis testing for model transformations. In A. Rensink & J. Warmer (Eds.), *Model driven architecture – foundations and applications* (pp. 376–390). Springer Berlin Heidelberg.

Papadakis, M., Kintis, M., Zhang, J., Jia, Y., Traon, Y. L., & Harman, M. (2019). Chapter six - mutation testing advances: An analysis and survey. *Advances in Computers*, *112*, 275–378.

Sánchez Cuadrado, J., Guerra, E., & de Lara, J. (2017). Static analysis of model transformations. *IEEE Trans. Software Eng.*, *43*(9), 868–897.

Sánchez Cuadrado, J., Guerra, E., de Lara, J., Clarisó, R., &

Cabot, J. (2017). Translating target to source constraints in model-to-model transformations. In *MODELS* (pp. 12–22). IEEE Computer Society.

Sirius. (2017). `https://eclipse.org/sirius/`.

Steinberg, D., Budinsky, F., Paternostro, M., & Merks, E. (2008). *EMF: Eclipse Modeling Framework, 2nd edition*. Addison-Wesley Professional.

Strug, J. (2016). Applying mutation testing for assessing test suites quality at model level. In *Fedcsis* (Vol. 8, pp. 1593–1596). IEEE.

Troya, J., Bergmayr, A., Burgueño, L., & Wimmer, M. (2015). Towards systematic mutations for and with ATL model transformations. In *ICST workshops* (pp. 1–10).

Ujhelyi, Z., Horváth, Á., & Varró, D. (2011). Static type checking of model transformation programs. *ECEASST*, *38*.

UML 2.5.1. Specification by the OMG. (2017). `https://www.omg.org/spec/UML/About-UML/`.

van Amstel, M., & van den Brand, M. G. J. (2011). Model transformation analysis: Staying ahead of the maintenance nightmare. In *Icmt* (Vol. 6707). Springer.

Zeller, A., Gopinath, R., Böhme, M., Fraser, G., & Holler, C. (2019). Mutation-based fuzzing. In *The fuzzing book*. Saarland University. https://www.fuzzingbook.org/html/MutationFuzzer.html. Retrieved from https://www.fuzzingbook.org/html/MutationFuzzer.html (Retrieved Oct 2019)

A. Mutation Operators for UML Class Diagrams

This appendix contains the definition the UML Class Diagram mutation operators proposed in (Granda et al. 2016) using WODEL.

UML CD mutation	WODEL code
Adds a redundant constraint to the CD	cd = **select one** ClassDiagram **where** {^constraints <> **null**} c = **select one** Constraint **in** cd→^constraints **deep clone one** c **in** cd→^constraints
Adds an extraneous constraint to the CD	**create** Constraint **with** {type = **one** Class, id = **random**−**string**(1, 4)}
Adds a redundant association to the CD	cd = **select one** ClassDiagram **where** {relations **is typed** Association} a = **select one** Association **in** cd→relations **deep clone** a **in** cd→relations
Adds a redundant derived association to the CD	cd = **select one** ClassDiagram **where** {relations **is typed** Association} a1 = **select one** Association **in** cd→relations a2 = **select one** Association **in** cd→relations **where** {src = a1→tar} **create** Association **in** cd→relations **with** { name = **random**−**string**(1, 4), src = a1→src, tar = a2→tar, derived = true}

Continued on next column...

...continued from previous column

UML CD mutation	WODEL code
Adds an extraneous association to the CD	**create** Association **with** {src = **one** Class, tar = **one** Class, name = **random**−**string**(1, 4)}
Adds an extraneous derived association to the CD	**create** Association **with** {src = **one** Class, tar = **one** Class, name = **random**−**string**(1, 4), derived = true}
Adds a redundant generalization to the CD	c = **select one** Class **where** {super <> **null and** super→super <> **null**} s1 = **select one** Class **in** cl→super **where** {super <> **null**} s2 = **select one** Class **in** s1→super **where** {**self** <> c→super} **modify** c **with** {super = s2}
Adds an extraneous generalization to the CD	c1 = **select one** Class **where** {super <> **null**} s = **select all** Class **in closure**(c1→super) c2 = **select one** Class **where** {**self** <> s} **modify** c1 **with** {super = c2}
Adds a redundant class to the CD	cd = **select one** ClassDiagram **where** {classes **is typed** Class} c = **select one** Class **in** cd→classes **deep clone** c **in** cd→classes
Adds an extraneous class to the CD	**create** Class **with** {name = **random**−**string**(1, 4)}
Adds a redundant association class to the CD	cd = **select one** ClassDiagram **where** {classes **is typed** AssociationClass} ac = **select one** AssociationClass **in** cd→classes **clone one** ac **in** cd→classes
Adds an extraneous association class to the CD	**create** AssociationClass **with** {name = **random**−**string**(1, 4)}
Adds a redundant attribute to a class	c = **select one** Class **where** {features **is typed** Attribute} att = **select one** Attribute **in** c→features **deep clone** att **in** c→features
Adds an extraneous attribute to a class	**create** Attribute **with** {type = **one** Class, name = **random**−**string**(1, 4)}
Adds a redundant operation to a class	c = **select one** Class **where** {features **is typed** Operation} op = **select one** Operation **in** c→features **deep clone** op **in** c→features
Adds an extraneous operation to a class	**create** Operation **with** {type = **one** Class, name = **random**−**string**(1, 4)}
Adds a redundant parameter to an operation	op = **select one** Operation **where** {params <> **null**} p = **select one** Parameter **in** op→params **clone** p **in** op→params
Adds an extraneous parameter to an operation	**create** Parameter **with** {name = **random**−**string**(1, 4)}
Changes a constraint by deleting the references to a class attribute	**remove one** PathElementCS **where** {pathName = **one** Attribute}
Changes attribute data type in a constraint	p = **select one** PathElementCS **where** {pathName = **one** Attribute} a = **select one** Attribute **in** path→pathName **modify** a **with** {type = **other** Class}
Changes a constraint by deleting the calls to specific operation	**remove one** OperationCS

Continued on next column...

...continued from previous column

UML CD mutation	WODEL code
Changes an arithmetic operator for another and supports binary operators: add, subtract, multiply, divide	**modify one** IntLiteralExpCS **with** {op **in** ['add', 'subtract', 'multiply', 'divide']}
Changes a constraint by adding the conditional operator 'not'	inv = **select one** InvariantCS **where** {exp <> **null**} **create** LogicExpCS **in** inv→exp **with** {op = 'not'}
Changes a conditional operator 'and' for a conditional operator 'or'	**modify one** BooleanExpCS **where** {op = 'and'} **with** {op = 'or'}
Changes a conditional operator 'or' for a conditional operator 'and'	**modify one** BooleanExpCS **where** {op = 'or'} **with** {op = 'and'}
Changes a constraint by deleting the conditional operator 'not'	**remove one** LogicExpCS **where** {op = 'not'}
Changes a relational operator for a different one	**modify one** BooleanExpCS **where** {op **in** ['lt', 'lte', 'gt', 'gte', 'equals', 'distinct']} **with** {op **in** ['lt', 'lte', 'gt', 'gte', 'equals', 'distinct']}
Changes a constraint by deleting a unary arithmetic operator 'negative'	**modify one** IntLiteralExpCS **where** {op = 'negative'} **with** {op = ''}
Interchanges the members (src and tar) of an association	**modify one** Association **with** {**swap**(src, tar)}
Changes the association type	**retype** **one** [Aggregation, Association, Composition, Dependency] **as** [Aggregation, Association, Composition, Dependency]
Changes the tar multiplicity of an association	**modify one** Association **with** {ntar **in** ['*', '0..1', '1..1', '1..*']}
Changes the generalization member ends	**modify one** Class **with** {super = **other** Class}
Changes visibility kind of the class	**modify one** Class **with** {visitibility **in** {public, private, protected, package}}
Changes class by an association class	**retype one** Class **as** AssociationClass **with** {association = **one** Association}
Changes association class for a class	**retype one** AssociationClass **as** Class
Changes the class feature 'abstract' to true	**modify one** Class **where** {abstract = false} **with** {abstract = true}

Continued on next column...

...concluded from previous column

UML CD mutation	WODEL code
Changes the attribute feature 'derived' to true	**modify one** Attribute **where** {derived = false} **with** {derived = true}
Changes the attribute property 'derived' to false	**modify one** Attribute **where** {derived = true} **with** {derived = false}
Changes the attribute data type	**modify one** Attribute **with** {type = **other** NamedElement}
Changes the attribute visibility property	**modify one** Attribute **with** {visibility **in** {public, private, protected, package}}
Changes the order of the parameters	op = **select one** Operation **where** {params <> **null**} p = **select one** Parameter **in** op→params **modify** op **with** {params −= p, params += p}
Changes the visibility kind of an operation	**modify one** Operation **with** {visibility **in** {public, private, protected, package}}
Changes the data type returned by an operation	**modify one** Operation **with** {type = **other** NamedElement}
Changes the parameter data type	**modify one** Parameter **with** {type = **other** NamedElement}
Deletes a constraint	**remove one** Constraint
Deletes an association	**remove one** Association
Deletes a generalization relation	c = **select one** Class **where** {super <> **null**} **remove one** Class **from** c→super
Deletes a class	c = **select one** Class **remove all** Relation **where** {src = c or tar = c} **remove all** [Operation, Feature] **where** {type = c} **remove all** AssociationClass **where** {association = **null**} **remove** c
Deletes an attribute	**remove one** Attribute
Deletes an operation	**remove one** Operation
Deletes a parameter from an operation	**remove one** Parameter

About the authors

Pablo Gómez-Abajo is Assistant Professor at the computer science department of the Universidad Autónoma de Madrid. After around 8 years working in the industry, he returned to Academia and joined the modelling and software engineering research group in 2015. Contact him at pablo.gomeza@uam.es, or visit https://www.gomezabajo.es.

Esther Guerra is Associate Professor at the computer science department of the Universidad Autónoma de Madrid. Together with J. de Lara, she leads the modelling and software engineering research group (http://miso.es). She is interested in flexible modelling, meta-modelling, domain specific languages and model transformation. Contact her at esther.guerra@uam.es, or visit http://www.eps.uam.es/~eguerra.

Juan de Lara is Full professor at the computer science department of the Universidad Autónoma de Madrid. Together with E. Guerra, he leads the modelling and software engineering research group. His research interests are in model-driven engi-

neering and automated software development. Contact him at juan.delara@uam.es, or visit http://www.eps.uam.es/~jlara/.

Mercedes G. Merayo holds an Associate Professor position in the Computer Systems and Computation Department of the Universidad Complutense de Madrid. She leads, together with Manuel Núñez, the Design and Testing of Reliable Systems research group. Her current research interests include model based testing, distributed testing, asynchronous testing, mutation testing and timed extensions in formal testing. Contact her at mgmerayo@fdi.ucm.es, or visit http://antares.sip.ucm.es/mercedes/.