Seed Model Synthesis for Testing Model-based Mutation Operators

Pablo Gómez-Abajo[1], Esther Guerra[1], Juan de Lara[1], and Mercedes G. Merayo[2]

[1] Modelling and Software Engineering group
http://miso.es
Universidad Autónoma de Madrid (Spain)
(Pablo.GomezA,Esther.Guerra,Juan.deLara)@uam.es
[2] Design and Testing of Reliable Systems group
http://antares.sip.ucm.es/testing/
Universidad Complutense de Madrid (Spain)
mgmerayo@fdi.ucm.es

Abstract. In 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 activities normally require the definition of 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, in previous work we proposed a model-based approach to create and execute mutation operators. Our proposal represents the artefacts to be mutated as models and provides a domain-specific language called Wodel to define the mutation operators. However, testing the operators is cumbersome, since it requires the manual creation of input seed models. To facilitate this testing process, we propose a method – based on model finding – for the automated synthesis of test models that exercise the defined mutation operators. We provide tool support for our proposal, and illustrate its usage by defining mutation operators for BPMN.

Keywords: Model-based mutation · Model-driven engineering · Model synthesis · OCL · Wodel · BPMN.

1 Introduction

Mutation consists in the selective introduction of modifications into sets of seed artefacts, like models, programs or data. Mutation is at the core of many techniques in software engineering, like mutation testing (where programs are mutated with faults to evaluate the quality of a test suite) [5,14], test data generation (like in mutation-based fuzzing, which introduces small changes to existing test inputs) [26], and search-based software engineering (which applies meta-heuristic search techniques to software engineering problems, where candidate

solutions are combined and mutated) [15]. Mutation has also been applied for other purposes, like the automatic generation of exercises and quizzes [9] or testing distributed applications in simulated environments [3].

Mutation-based methods require the creation of mutation operators able to change the target artefacts in pertinent ways. For example, for mutation testing, operators need to emulate common faults made by competent developers. 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, operators are often defined ad-hoc using general programming languages not designed for mutating artefacts, like Java [19] or C [18], which is costly and error-prone.

To improve this situation, we propose an approach to facilitate the creation and testing of mutation operators. It 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 [8,9] specially tailored to design mutation operators applicable over models. To help in the validation of the designed operators, we offer facilities for synthesizing models over which the operators can be tested. Such models are ensured to provide full statement coverage of the mutation program.

Our method is supported by the Wodel tool [10]. While the DSL Wodel was introduced in [8,9], in this paper we focus on the facility for seed model synthesis, based on constraint solving and model finding [20].

The rest of this paper is organized as follows. Section 2 introduces a running example in the area of process modelling. Then, Section 3 describes the Wodel DSL. Section 4 explains our methods to synthesize models for testing mutation operators, and Section 5 describes our current tool support. Finally, Section 6 reviews related work, and Section 7 concludes the paper.

2 Running Example: Mutation for Process Models

A number of research works have applied mutation to workflow languages for different purposes, like evaluating the quality of test cases (as in [7] for WS-BPEL), optimising process models (as in [16] for BPMN), or for process mining using a genetic approach (as in [6] for process trees or [22] for Petri nets). As an illustration, in this paper, we are defining a set of mutation operators for the Business Process Model and Notation (BPMN)[3].

Figure 1 shows part of a simplified BPMN meta-model taken from an editor built by a third-party[4]. A process defines a set of FlowObjects, which can be either Activities (i.e., a work to be done), Events (to denote that something happens, such as the start or end of the process), or Gateways (to fork or merge several paths). Depending on the kind of gateway, the execution of its outgoing paths can be

[3] http://www.bpmn.org/

[4] https://github.com/bluezio/simplified-bpmn-example

in parallel (AND), inclusive (OR, one or more paths are executed), or exclusive (XOR, only a path is executed). The OCL invariant inv2 ensures that gateways always have input and output paths. Finally, flow objects can be connected through ConnectingObjects to specify the execution flow (Sequence), send messages (Message), or associate artefacts to flow objects (Association). The OCL invariant inv1 ensures that start events have no input flows, end events have no output flows, and flow objects are not connected to themselves. To better illustrate model synthesis in Section 4, we have restricted processes to have between 1 and 10 elements, as cardinality of reference BusinessProcess.elements indicates.

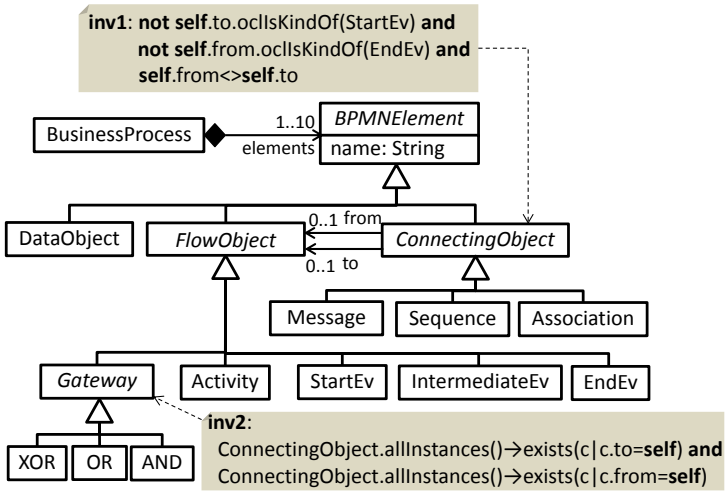

Fig. 1: Simplified BPMN meta-model.

Figure 2 shows a simple BPMN model in concrete syntax. It describes the process to satisfy someone who is hungry. The process starts when a person becomes hungry. The first activity is to buy food, followed by cooking the food. Then, when the meal is ready, the person eats it, and this concludes the process.

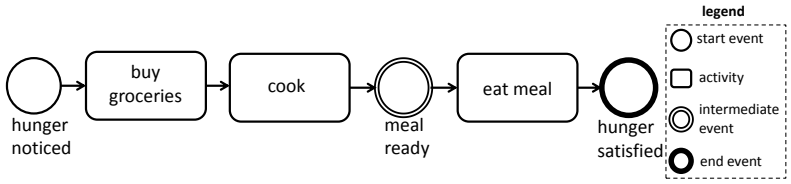

Fig. 2: An example BPMN model (using the standard concrete syntax).

In the next section, we introduce our DSL Wodel for model mutation, and use it to define a set of mutation operators for BPMN.

3 Wodel: a Domain Specific Language for Model Mutation

Wodel [8,9] is a DSL for the specification of mutation operators. It is domain-independent, and so it can be applied to arbitrary languages, 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. The execution of a Wodel program yields a set of mutant models obtained by applying the specified operators to a set of given seed models, using different policies. For traceability, a registry with the mutations used to generate each mutant is also produced. Wodel ensures that the created mutant models conform to the domain meta-model and satisfy its OCL invariants.

Wodel provides mutation primitives to *select*, *modify*, *create*, *delete*, *clone* and *retype* objects; and to *create*, *modify* and *delete* references. 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. Its editing environment [10] features code completion, type checking, and generation of stand-alone Java code from Wodel programs (cf. Section 5). The tool can be extended with post-processing applications. Two examples are the framework for the automated generation of exercises presented in [9], and the mutation testing development framework introduced in [11].

Listing 1 shows a simple Wodel program for defining a mutation operator for BPMN. Line 1 specifies the strategy for mutant synthesis: generating either a maximum number of mutants, or all possible ones by using the keyword exhaustive. Line 2 states the output folder to store the mutants, and the input folder with the seed models. Line 3 configures the meta-model in use (we use the on in Fig. 2). The remainder of the program defines the mutation operators.

```
1  generate exhaustive mutants
2  in "out/" from "model/"
3  metamodel "http://bpmn.com"
4
5  with blocks {
6    ev2ac {
7      retype
8        one [StartEv, IntermediateEv, EndEv]
9        as Activity
10   }
11 }
```

Listing 1: Defining a mutation operator for BPMN with Wodel.

In this example, the operator ev2ac retypes an event of any kind into an activity (lines 7–9). This operator uses a single mutation primitive, but in general, operators can use any number of mutation statements. For instance, Table 2 in the appendix contains other more complex operators for BPMN, both proposed in the literature [21] and created by us. 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 just once.

Fig. 3 shows an application of the mutation operator in Listing 1 to the BPMN model of Fig. 2. In the resulting mutant, the IntermediateEv 'meal ready' is replaced by an equally named Activity, and the incoming/outgoing references are preserved by the operator.

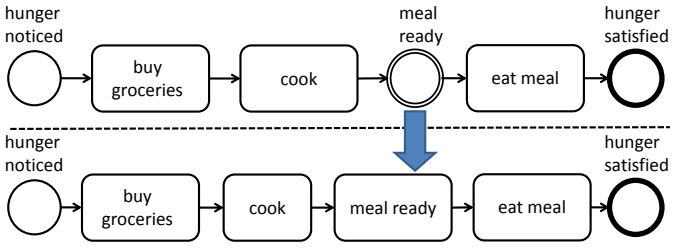

Fig. 3: Application of the mutation operator in Listing 1 to a BPMN model.

4 Seed Model Synthesis using Model Finding

As any other software, mutation programs need be tested to detect possible errors, so that they can be fixed. In the case of Wodel, this implies the creation of test models upon which the mutation programs can be executed. However, creating test models manually is tedious and error-prone, and it is difficult to ensure a full coverage of the program.

Therefore, to ease the testing of Wodel programs and operators, we propose a method based on model finding [17] to automatically produce seed models over which all instructions of the given Wodel program are applicable (if such models exist in the given search scope).

Figure 4 outlines the seed model synthesis process. It relies on model search, a technique which applies constraint resolution over models [17]. 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 [20], 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 1 shows the templates used to generate the OCL constraints for each mutation primitive, as well as 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 takes place.

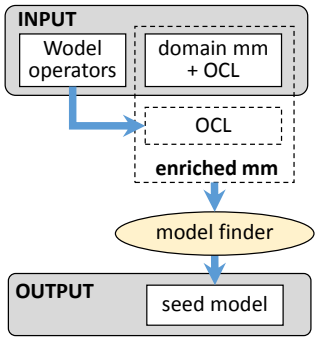

Fig. 4: Process for automated model synthesis.

The table shows as an example the deletion of an Activity: the derived OCL constraint checks that there exists some Activity, and the BusinessProcesses to which it belongs contain other elements apart from the Activity (i.e., the size of reference BusinessProcess.elements is bigger than 1, and deleting the Activity 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 reasons, Table 1 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 Activity has to occur twice, we generate the constraint shown in Listing 2 (cf. Table 1).

```
BusinessProcess.allInstances()→exists(b1,b2 |
  (b1 <> b2 and b1.elements→size() < 10 and b2.elements→size() < 10)
  or b1.elements→size() < 9)
```

Listing 2: OCL invariant derived from a Wodel program creating two activities.

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** Activity **where** {name = 'InitialName'} **with** {name = 'ModifiedName'} **OCL:** Activity.**allInstances**() →**exists**(a | a.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:** a = **create** Activity **OCL:** BusinessProcess.**allInstances**() →**exists**(b | b.elements→**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** Activity **OCL:** Activity.**allInstances**()→**exists**(a | BusinessProcess.**allInstances**() →**forAll**(b | b.elements→**includes**(a) **implies** b.elements→**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** Sequence **OCL:** Sequence.**allInstances**() →**exists**(s | true) **and** BusinessProcess.**allInstances**() →**exists**(b | b.elements→**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). Orcatenate 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** Activity **as** DataObject **OCL:** Activity.**allInstances**() →**exists**(a | ConnectingObject.**allInstances**() →**forAll**(c | (c.to→**includes**(a) **implies** c.to→**size**()>0) **and** (c.from→**includes**(a) **implies** c.from→**size**()>0))) −− *the checks on references to* −− *and from are performed because* −− *they are not compatible with* −− *DataObjects*
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** ^to **to one** Activity **in one** Sequence **OCL:** Activity.**allInstances**()→**exists**(a | true) **and** Sequence.**allInstances**() →**exists**(s | s.to→**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** ^to **from one** Sequence **to other** FlowObject **OCL:** Sequence.**allInstances**() →**exists**(s | s.to→**notEmpty**()) **and** FlowObject.**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:** a = **select one** Sequence **remove** a→^to **OCL:** Sequence.**allInstances**() →**exists**(s | s.to→**size**() > 0)

Table 1: Templates to generate OCL constraints from mutation primitives.

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 1 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 3 shows the OCL constraint generated from the program in Listing 1. As the retype operation considers three types, and or with three cases is generated.

```
1 context Dummy
2
3 inv mut1 :
4  StartEv.allInstances()→exists(a | true) or
5  IntermediateEv.allInstances()→exists(a | true) or
6  EndEv.allInstances()→exists(a | true)
```

Listing 3: OCL constraint derived from Listing 1.

Figure 5 shows a seed model returned by the model finder for the previous constraint. It satisfies the constraint of Listing 3, and those of the original meta-model.

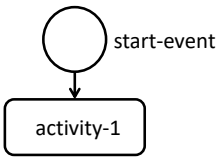

Fig. 5: Generated seed model.

Please note that seed models satisfying the synthesized constraints 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 existing invariants of the domain meta-model. This would require from techniques for advancing constraints to model operations [4], which is left for future work.

5 Tool Support

The Wodel development environment is available as an Eclipse plugin at `http://miso.es/tools/Wodel.html`, together with examples and videos. The implementation is based on EMF [24], and expects the meta-models of the artefacts to be mutated to be specified using Ecore.

Figure 6 shows the Wodel IDE. The IDE features a textual editor (label 1 in the figure) to create Wodel programs. The editor is built with Xtext and supports features like code completion. Label 2 in the figure shows the explorer with a typical Wodel organization. The src folder contains Wodel programs with the defined mutation operators. These operators are compiled into Java programs, and stored in the src-gen folder. The generated Java programs can be executed within the IDE to produce mutants from the seed models, which are saved in the out folder.

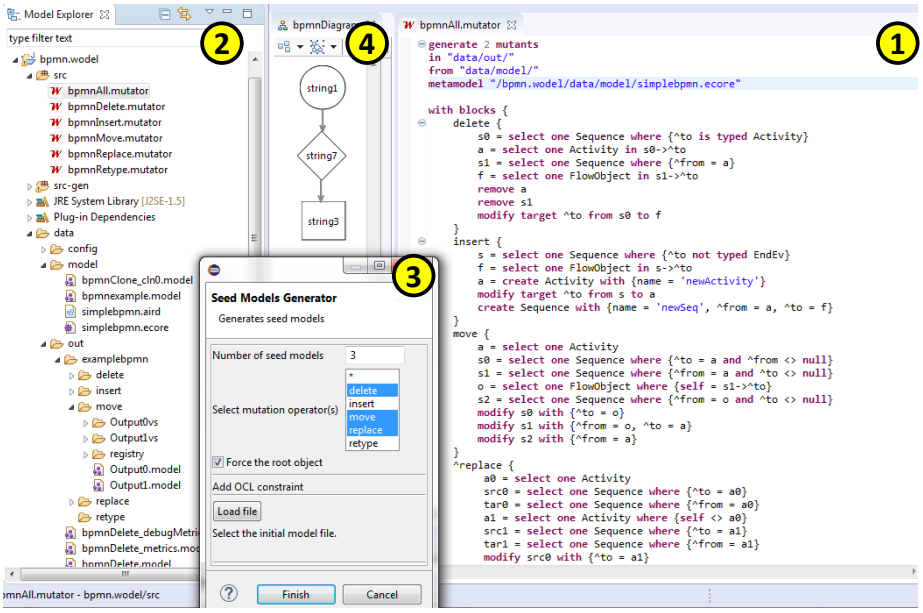

Fig. 6: Wodel IDE and seed model synthesizer.

To support the contributions presented in this paper, we have extended the Wodel environment to support the synthesis of seed models for testing mutation programs. The seed model synthesis for a given program can be configured by means of the wizard marked with label 3 in the figure. This wizard 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 using the Use Validator model finder [20]. The generated seed models are converted from the USE format to EMF, and stored in the model folder (see the explorer view). The generated models can be used to test the designed operators, and can be visualized using, e.g., the EMF tree editor, or dedicated graphical editors, such as the one with label 4 in the figure.

6 Related Work

Next, we review works related to the main elements of our approach: languages tailored to define or synthesize mutation operators, and model synthesis from requirements.

DSLs for mutation operators, and operator synthesis. Some model-based mutation approaches use general-purpose model transformation languages to define mutation operators. In [12], the authors present an MDE approach to define mutation testing tools, where programs are represented as models, and operators are encoded in QVT-o. Mutation operators have also been defined using Henshin in [2], and ATL in [25]. 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 [25] miss such policies and only produce one mutant per input model.

Major [19] 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.

In [1], 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 [2] generates operators that guarantee the consistency of the mutated models with the meta-model multiplicity constraints. The operators are encoded as graph transformation rules. In comparison, Wodel 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 [20] or Alloy [17]) for activities like model completion, test model generation, or transformation analysis. For example, model finding is used in [13] to generate test models for transformations based on specifications. For this purpose, the specifications are transformed into OCL. In our case, the novelty yields in the encoding of the semantics of the Wodel program into OCL, ensuring full statement coverage of the program.

Overall, to the best of our knowledge, environments to support the creation and testing of model-based mutation operators are currently lacking. Hence, we have designed Wodel, and its seed model generation capabilities to fill this gap.

7 Conclusions and Future Work

Given the recurrent need to develop sets of mutation operators, we propose a model-based approach to facilitate their definition, testing, and application. This way, we provide a DSL – called Wodel– for their description, and model synthesis capabilities – based on model finding – for their testing and validation. Our approach is supported by an Eclipse plugin, and we have illustrated the approach in the context of the BPMN language.

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 [23]. Second, to generate seed models ensuring that the execution of the Wodel program leads to a correct model. For this purpose, we may use techniques to advance OCL constraints as preconditions, based on [4]. We also plan to work on static analysis techniques, e.g., to detect operator conflicts and dependencies. Finally, we are currently working on a methodology supporting the integral engineering of mutation operators.

Acknowledgments. Work funded by the Spanish Ministry of Science (projects MASSIVE, RTI2018-095255-B-I00 and FAME, RTI2018-093608-B-C31) and the R&D programme of Madrid (project FORTE, P2018/TCS-4314).

References

1. Alhwikem, F., Paige, R.F., Rose, L., Alexander, R.: A systematic approach for designing mutation operators for MDE languages. In: MODEVA. CEUR Workshop Proceedings, vol. 1713, pp. 54–59. CEUR-WS.org (2016)
2. Burdusel, A., Zschaler, S., John, S.: Automatic generation of atomic consistency preserving search operators for search-based model engineering. In: MODELS. pp. 106–116. IEEE (2019)
3. Cañizares, P.C., Núñez, A., Merayo, M.G.: Mutomvo: Mutation testing framework for simulated cloud and hpc environments. Journal of Systems and Software **143**, 187 – 207 (2018)
4. Cuadrado, J.S., Guerra, E., de Lara, J., Clarisó, R., Cabot, J.: Translating target to source constraints in model-to-model transformations. In: MODELS. pp. 12–22. IEEE Computer Society (2017)
5. DeMillo, R.A., Lipton, R.J., Sayward, F.G.: Hints on test data selection: Help for the practicing programmer. IEEE Computer **11**(4), 34–41 (1978)
6. van Eck, M.L., Buijs, J.C.A.M., van Dongen, B.F.: Genetic process mining: Alignment-based process model mutation. In: BPM workshops. LNBIP, vol. 202, pp. 291–303. Springer (2014)
7. Estero-Botaro, A., Palomo-Lozano, F., Medina-Bulo, I., Domínguez-Jiménez, J.J., García-Domínguez, A.: Quality metrics for mutation testing with applications to WS-BPEL compositions. Softw. Test., Verif. Reliab. **25**(5-7), 536–571 (2015)
8. Gómez-Abajo, P., Guerra, E., de Lara, J.: Wodel: a domain-specific language for model mutation. In: SAC. pp. 1968–1973. ACM (2016)
9. Gómez-Abajo, P., Guerra, E., de Lara, J.: A domain-specific language for model mutation and its application to the automated generation of exercises. Computer Languages, Systems & Structures **49**, 152 – 173 (2017)
10. Gómez-Abajo, P., Guerra, E., de Lara, J., Merayo, M.G.: A tool for domain-independent model mutation. Sci. Comput. Program. **163**, 85–92 (2018)
11. Gómez-Abajo, P., Guerra, E., de Lara, J., Merayo, M.G.: Mutation testing for DSLs (tool demo). In: DSM. pp. 60–62. ACM (2019)
12. González, A., Luna, C., Bressan, G.: Mutation testing for Java based on model-driven development (in spanish). In: CLEI-SLISW (2018)
13. Guerra, E., Soeken, M.: Specification-driven model transformation testing. Software and Systems Modeling **14**(2), 623–644 (2015)
14. Hamlet, R.G.: Testing programs with the aid of a compiler. IEEE Trans. Software Eng. **3**(4), 279–290 (1977)

15. Harman, M., Jones, B.F.: Search-based software engineering. Information & Software Technology **43**(14), 833–839 (2001)
16. Herbert, L., Hansen, Z., Jacobsen, P., Cunha, P.: Evolutionary optimization of production materials workflow processes. Procedia CIRP **25**, 53–60 (2014)
17. Jackson, D.: Alloy: a language and tool for exploring software designs. Commun. ACM **62**(9), 66–76 (2019)
18. Jia, Y., Harman, M.: MILU: A customizable, runtime-optimized higher order mutation testing tool for the full C language. In: TAICPART. pp. 94–98 (2008)
19. Just, R.: The Major mutation framework: Efficient and scalable mutation analysis for Java. In: ISSTA. pp. 433–436. ACM (2014)
20. Kuhlmann, M., Gogolla, M.: From UML and OCL to relational logic and back. In: MODELS. LNCS, vol. 7590, pp. 415–431. Springer (2012)
21. Li, C., Reichert, M., Wombacher, A.: On measuring process model similarity based on high-level change operations. In: ER. LNCS, vol. 5231, pp. 248–264. Springer (2008)
22. de Medeiros, A., Weijters, A., van der Aalst, W.: Genetic process mining: an experimental evaluation. Data Min. Knowl. Discov. **14**(2), 245–304 (2007)
23. Montaghami, V., Rayside, D.: Bordeaux: A tool for thinking outside the box. In: FASE. LNCS, vol. 10202, pp. 22–39. Springer (2017)
24. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework, 2^{nd} Edition. Addison-Wesley Professional (2008)
25. Troya, J., Bergmayr, A., Burgueño, L., Wimmer, M.: Towards systematic mutations for and with ATL model transformations. In: ICST Workshops. pp. 1–10 (2015)
26. Zeller, A., Gopinath, R., Böhme, M., Fraser, G., Holler, C.: Mutation-based fuzzing. In: The Fuzzing Book. Saarland University (2019), `https://www.fuzzingbook.org/html/MutationFuzzer.html`, retrieved Oct 2019

A BPMN Mutation Operators

Table 2 encodes the BPMN mutation operators proposed in [21] using Wodel.

Mutation operator	**Wodel code**
Insert activity	s = **select one** Sequence **where** {^to not **typed** EndEv} f = **select one** FlowObject **in** s0→^to a = **create** Activity **with** {name = 'newActivity'} **modify target** ^to **from** s **to** a **create** Sequence **with** {name = 'newSeq', ^from = a, ^to = f}
Remove activity	s0 = **select one** Sequence **where** {^to **is typed** Activity} a = **select one** Activity **in** s0→^to s1 = **select one** Sequence **where** {^from = a} f = **select one** FlowObject **in** s1→^to **remove** a, s1 **modify target** ^to **from** s0 **to** f

Move activity	a = **select one** Activity
	s0 = **select one** Sequence **where** {^to = a **and** ^from <> **null**}
	s1 = **select one** Sequence **where** {^from = a **and** ^to <> **null**}
	o = **select one** FlowObject **where** {**self** = s1→^to}
	s2 = **select one** Sequence **where** {^from = o **and** ^to <> **null**}
	modify s0 **with** {^to = o},
	  s1 **with** {^from = o, ^to = a},
	  s2 **with** {^from = a}

Mutation operator	**Wodel code**
Replace activity	a0 = **select one** Activity src0 = **select one** Sequence **where** {^to = a0} tar0 = **select one** Sequence **where** {^from = a0} a1 = **select one** Activity **where** {**self** <> a0} src1 = **select one** Sequence **where** {^to = a1} tar1 = **select one** Sequence **where** {^from = a1} **modify** src0 **with** {^to = a1}, tar0 **with** {^from = a1}, src1 **with** {^to = a0}, tar1 **with** {^from = a0}
Retype gateway	**retype one** [AND, OR, XOR] **as** [AND, OR, XOR]

Table 2: Wodel mutation operators for BPMN.