Wodel: A Domain-Specific Language for Model Mutation

Pablo Gómez-Abajo, Esther Guerra, Juan de Lara
Computer Science Department
Universidad Autónoma de Madrid
{Pablo.GomezA, Esther.Guerra, Juan.deLara}@uam.es

ABSTRACT

Model-Driven Engineering (MDE) is a software engineering paradigm that uses models as main assets in all development phases. While many languages for model manipulation exist (e.g., for model transformation or code generation), there is a lack of frameworks to define and apply model mutations.

A model mutant is a variation of an original model, created by specific model mutation operations. Model mutation has many applications, for instance, in the areas of model transformation testing, model-based testing or education.

In this paper, we present a domain-specific language, called Wodel, for the specification and generation of model mutants. Wodel is domain-independent, as it can be used to generate mutants of models conforming to arbitrary meta-models. Its development environment is extensible, permitting the incorporation of post-processors for different applications. As an example, we show an application consisting on the automated generation of exercises for particular domains (automata, class diagrams, electronic circuits, etc.).

CCS Concepts

•Software and its engineering → Domain specific languages; Specialized application languages; Source code generation;

Keywords

Model-Driven Engineering; Domain-Specific Languages; Model Mutation; Education

1. INTRODUCTION

Model-Driven Engineering (MDE) [3] uses models in all phases of the software development process, where they are used to specify, simulate, test and generate code for the final system. Hence, model manipulation is a key activity in MDE, for which domain-specific languages (DSLs) are heavily used. For example, many DSLs exist to simulate models, produce a model from another one, or synthesize code.

A *model mutation* is a kind of model manipulation that creates a set of variants (or *mutants*) of a seed model by the application of one or more *mutation operators*. Model mutation has many applications. For example, in model transformation testing [1], a transformation is represented as a model that is mutated to evaluate the efficacy of a test model set. Such a test set may have been created by mutation of a set of input seed models. In education, a model representing a correct solution in a domain (like a class diagram, an automaton or an electronic circuit) is mutated to produce exercises that can be automatically graded [9].

There are some frameworks for model mutation, but they are specific for a language (e.g., logic formulae [5]) or domain (e.g., testing [1, 2]); moreover, mutation operators are normally created using general-purpose programming languages not tailored to the definition and production of mutants. Hence, there is a lack of proposals facilitating the definition of mutation operators, applicable to arbitrary languages and applications. These would facilitate the creation of domain-specific mutation frameworks like the abovementioned ones by providing: high-level mutation primitives (e.g., for object creation or reference redirection) together with strategies for their customization; support for composition of mutation operators; handy integration with external applications through compilation into a general-purpose language; and traceability of the applied mutations.

To facilitate the specification and creation of model mutations in a meta-model independent way, we propose a DSL called Wodel. This provides primitives for model mutation (e.g., creation, deletion, reference reversal), item selection strategies (e.g., random, specific, all), and composition of mutations. We have built a development environment which allows creating Wodel programs and their compilation into Java, and can be extended with post-processor steps for particular applications. We illustrate our approach by the automated generation of finite automata exercises.

Paper organization. Sect. 2 overviews our approach. Next, Sect. 3 presents Wodel, and Sect. 4 tool support. Sect. 5 applies Wodel to the generation of test exercises. Sect. 6 discusses related works, and Section 7 concludes the paper.

SAC 2016,April 04-08, 2016, Pisa, Italy
Copyright 2016 ACM 978-1-4503-3739-7/16/04. . . $15.00
http://dx.doi.org/10.1145/2851613.2851751

2. OVERVIEW AND RUNNING EXAMPLE

In MDE, models must conform to a meta-model which declares the admissible model elements, properties and relations. Thus, our goal is to make available a DSL to specify mutation operators and their application strategy for models conformant to a given arbitrary meta-model, and facilitate the use of the generated mutants for different applications.

Fig. 1 shows the workflow of our approach. First, the user provides a set of seed models conformant to a meta-model (label 1). Then, he uses Wodel to define the desired mutation operators and their execution details, like how many mutations of each type should be applied in each mutant, or their execution order (label 2). In addition, each Wodel program needs to declare the meta-model of the models to mutate, which can be any as Wodel is meta-model independent. This allows type-checking the program to ensure it only refers to valid meta-model types and properties, and allows checking that the result of the mutation is valid.

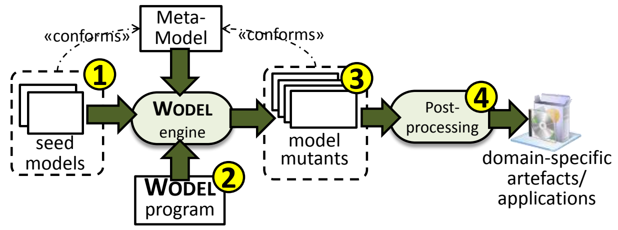


Figure 1: Scheme of our approach

Executing a Wodel program produces mutants of the seed models (label 3). These are still valid models (i.e., they conform to the seed models' meta-model) as this is checked upon generating each mutant. Finally, an optional post-processing step can be used to generate domain-specific artefacts for particular applications of the mutants (label 4).

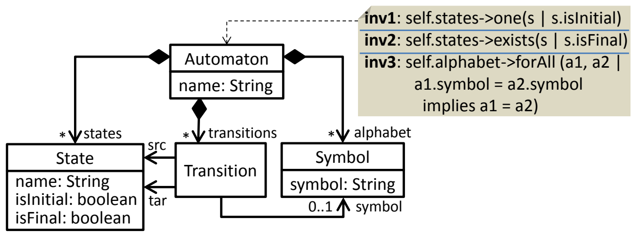
2.1 Running example

We will illustrate our proposal with an application of model mutation to education. In particular, we will generate exercises where students are presented a correct automaton (according to a specification) and other incorrect ones obtained by mutating the former, and students have to identify the correct one. Our approach permits generating exercises with different degrees of difficulty and automatic correction.

Fig. 2 shows the meta-model for automata used in the example. An Automaton is made of States, Transitions, and an alphabet of symbols. A State has a name and can be initial and/or final. A Transition connects two states and may have a symbol attached; if it lacks a symbol, it is a λ-transition. The meta-model includes three OCL invariants that any Automaton must fulfill: the first one demands exactly one initial state, the second one demands at least one final state, and the last requires distinct alphabet symbols.

3. THE WODEL DSL

In this section, we introduce our DSL Wodel and illustrate its usage showing examples of mutation operators for finite automata conformant to the meta-model in Fig. 2.



```
inv1: self.states->one(s | s.isInitial)
inv2: self.states->exists(s | s.isFinal)
inv3: self.alphabet->forAll (a1, a2 |
        a1.symbol = a2.symbol
         implies a1 = a2)
```

Figure 2: Meta-model for finite automata

Wodel programs have two parts. The first one declares the number of mutants to generate, output folder, seed models and their meta-model. The second part defines mutation operators and how many times they should be applied.

Listing 1 shows a simple Wodel program. Line 1 states that we want to generate 3 mutants in folder out, from the seed model evenBinary.fa. Line 2 indicates the meta-model of the seed model. Lines 4–9 define three mutation operators: the first one (lines 5–6) selects randomly a final state, and sets it to non-final; the second one (line 7) creates a new final state; and the last one (line 8) creates a new transition from the state selected in line 5 to the one created in line 7.

```
1  generate 3 mutants in "out/" from "evenBinary.fa"
2  metamodel "http://fa.com"
3
4  with commands {
5    s0 = modify one State where {isFinal = true}
6         with {reverse(isFinal)}
7    s1 = create State with {isFinal = true}
8    t0 = create Transition with {src = s0, tar = s1, symbol = one Symbol}
9  }
```

Listing 1: A simple Wodel program

Next, we detail the mutation primitives offered by Wodel. These include atomic operations to create and delete objects and references, modify attribute values, or redirect the source or target of references. Fig. 3 shows an excerpt of the Wodel meta-model with the definition of some representative mutation primitives. All mutation kinds inherit from Mutation, which holds the minimum and maximum number of times the mutation is to be applied. If this information is omitted, like in the mutations of Listing 1, they are executed once. In its turn, a Mutation is an ObjectEmitter which can receive a name, so that it can be referenced from other mutations. For example, in line 8 of Listing 1, the name s0 is used to refer to the State modified in line 5. The main supported kinds of Mutation are the following:

- CreateObject: It creates an object of the class indicated by the type reference. Optionally, it is possible to select a container object for the created one using an ObjectSelectionStrategy (explained below). In such a case, refType indicates the container's reference where the new object will be placed. If no container object is given, then Wodel selects a suitable one, and if several exist, one is chosen at random. In line 7 of Listing 1, it is not necessary to specify a container for the new State because, assuming one Automaton per model, the created state can only be placed in collection states of the automaton. Alternatively, we could make explicit the container object using **create**

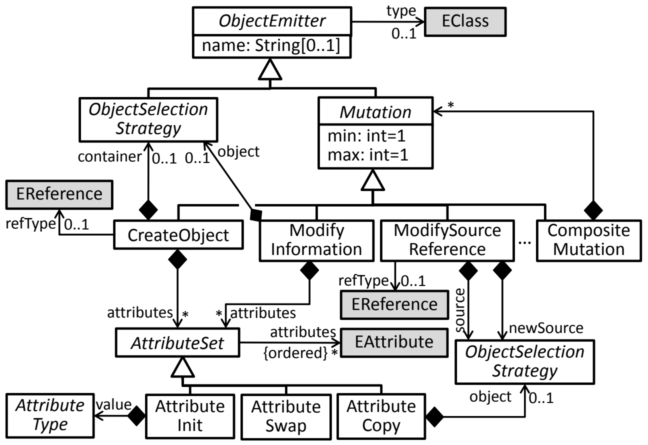


Figure 3: Some supported mutations

State **in one** Automaton.states. Similarly, it is possible to specify a value for the attributes and references of the new object, and in case no value is given for a mandatory reference, WODEL assigns it one object of a compatible type. Finally, note that shaded classes EClass, EReference and EAttribute belong to the meta-modelling framework used to build the domain meta-model (EMF [10] in our case). For instance, in our running example, Automaton is an EClass, states is an EReference, and name is an EAttribute. This way to refer to the domain meta-model elements enables type-checking and content assistance.

- CreateReference: It creates a new reference of the given type between two objects. The objects may be selected using an ObjectSelectionStrategy, or otherwise, source and target objects of a suitable type are chosen at random.
- ModifyInformation: It selects an object by means of an ObjectSelectionStrategy, and provides a set of modifications to be performed on its attributes (class AttributeSet). The meta-model shows just a few of the possible modifications, like initializing the value of an attribute, swapping the value of two attributes or references, and copying the value of one attribute to another. Other modifications depend on the attribute type. For example, it is possible to reverse the value of boolean attributes (as done in line 6 of Listing 1), while strings can be transformed into upper/lower case, be substituted by a random choice within a set, or some part of the string can be replaced.
- ModifySourceReference, ModifyTargetReference: It redirects the source or target of a reference to another object selected by an ObjectSelectionStrategy.
- RemoveObject: It safely removes an object selected by an ObjectSelectionStrategy, ensuring no dangling edge to/from the removed object remains.
- RemoveReference: It removes a reference of the given type. The source and target objects of the reference can be customised using ObjectSelectionStrategies.
- CompositeMutation: It allows defining composite mutations made of a sequence of atomic or other composite mutations, all of which are executed in a block.

Additionally, a Select operation permits selecting objects or references according to some criteria, so that they can be used in subsequent mutations.

Object and reference selection in mutations and selectors can be done using the following strategies: select a random element, a specific element (referenced by the name of an emitter), all elements satisfying some condition, or a different element to the one selected by the current mutation. The meta-model in Fig. 4 shows three of these strategies. SpecificObjectSelection selects an object referenced by an emitter. SpecificReferenceSelection selects both an object and a reference defined by the object's class. RandomObjectSelection chooses a random object from the class specified by reference type. All strategies can be parameterized with a condition (class Expression) on the attribute and reference values of the selected element. For example, in line 5 of Listing 1, the ModifyInformation mutation uses a RandomObjectSelection strategy (**one** State) with an attribute condition (**where** {isFinal = **true**}).

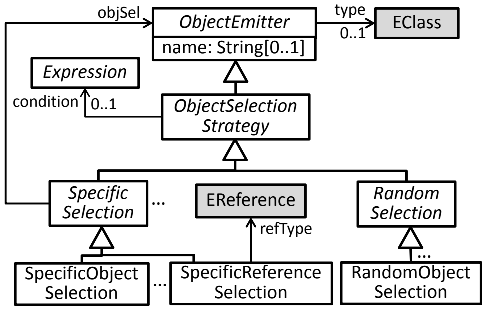


Figure 4: Some supported selection strategies

As we have seen, WODEL has a textual concrete syntax. Listing 2 shows a brief excerpt of its grammar.

```
WODELPROGRAM ::= DEFINITION with commands { MUTATION* }

DEFINITION ::=
  generate <num> mutants in <folder> from SEEDS
  metamodel <meta−model>

MUTATION ::=
  ( CREATEOBJECT | MODIFYINFORMATION |
   MODIFYSOURCEREFERENCE | ... | COMPOSITEMUTATION )
  ( [ (<min> ..)? <max> ] )?

CREATEOBJECT ::=
  ( <name> '=' )? create <EClass>
  ( in OBJECTSELECTIONSTRATEGY ( '.' <EReference> )? )?
  ( with { ATTRIBUTESET ( , ATTRIBUTESET )* } )?

MODIFYINFORMATION ::=
  ( <name> '=' )? modify OBJECTSELECTIONSTRATEGY
  with { ATTRIBUTESET ( , ATTRIBUTESET )* }

MODIFYSOURCEREFERENCE ::=
  modify source <EReference>
  ( from OBJECTSELECTIONSTRATEGY )?
  ( to OBJECTSELECTIONSTRATEGY )?

COMPOSITEMUTATION ::= ( <name> '=' )? [ MUTATION* ]
...
```

Listing 2: Excerpt of Wodel grammar

Listing 3 shows another example that generates 3 mutants from every model in folder models (line 1). Lines 5–7 define a composite mutation which removes a random non-initial State (line 5), and then removes all Transition objects pointing to, or stemming from, the deleted state (lines 6–7). The transitions to delete are those having an undefined value in references src or tar. The mutation in lines 8–10 selects a Transition randomly, and modifies its reference symbol to point to a different Symbol. The mutations define a cardinality, so that the first one will be applied between 0 and 2 times in every mutant, and the second one between 1 and 3.

```
 1 generate 3 mutants in "out/" from "models/" all
 2 metamodel "http://fa.com"
 3
 4 with commands {
 5   c0 = [ remove one State where {isInitial = false}
 6          remove all Transition where {src = null}
 7          remove all Transition where {tar = null} ] [0..2]
 8   modify target symbol
 9          from one Transition
10          to other Symbol [1..3]
11 }
```

Listing 3: Composite mutation and cardinalities

Since WODEL programs handle each defined mutation as an operation, it is not possible to have contradictory mutations; however, two mutations may cancel each other (e.g., one mutation creates an object, and another deletes it).

3.1 Expresiveness and succinctness of Wodel

Next, we discuss the expressivity and conciseness of WODEL. Expressivity is approached by using WODEL to define interesting mutations for automata, both devised by us and found in the literature [9]. Note that [9] does not consider final states or mutate transition symbols, but it applies different number of mutations which we express with cardinalities.

Table 1 lists the mutations. The first twelve change the language recognized by an automaton (assuming it is minimal), and the last two make an automaton non-deterministic. Although WODEL lacks the control structures of full-fledged programming languages, its expressivity was enough to express the mutations for our running example. Moreover, loops can be emulated with composite mutations and cardinalities, and conditionals are implicit in the conditions of selection strategies. While the expressivity of WODEL does not depend on the meta-model for which the mutations are defined, we acknowledge that its usage in other application contexts (e.g., model transformation testing) may require introducing new WODEL primitives.

We analyse conciseness by comparing with the equivalent Java code, which would be a natural alternative for integrating mutation operations into applications. Programming the mutation operations in Java would require from knowledge of the EMF reflective API [10], as one cannot assume that Java implementation classes exist for the types in the given meta-model. It also requires taking care of accidental details that WODEL manages for free, like placement of objects in containers, initialization of mandatory references, type-checking of mutations w.r.t. the meta-model, model serialization, checking well-formedness of resulting mutants, or comparing for equal resulting mutants.

To illustrate the complexity of the equivalent Java code,

Mutations that change the language	
Create transition [9]	**create** Transition **with** {symbol = **one** Symbol}
Create final state	Listing 1, line 7

Create connected state	s = **create** State
	with {name = **random−string**(1,4)}
	t = **create** Transition
	with {tar = s, symbol = **one** Symbol}

Delete transition	**remove one** Transition
Delete state and adjacent transitions	Listing 3, lines 5–7
Change symbol in transition	Listing 3, lines 8–10
Change final state to non-final	Listing 1, lines 5–6

Change initial state to a different one [9]	s0 = **modify one** State **where** {isInitial = true}
	with {isInitial = false}
	s1 = **modify one** State **where** {**self** <> s0}
	with {isInitial = true}

Swap direction of transition [9]	**modify one** Transition **with** {**swap**(src, tar)}

Swap symbol of two sibling transitions	t = **select one** Transition
	modify one Transition
	where {**self** <> t **and** src = t.src}
	with {**swap**(symbol, t.symbol)}

Redirect transition to a new final state	s = **create** State **with** {name='f', isFinal=true} **modify target** tar **from one** Transition **to** s

Combination of adding a new transition and changing the initial state [9]	s0 = **modify one** State **where** {isInitial = true}
	with {**reverse**(isInitial)}
	s1 = **modify one** State **where** {**self** <> s0}
	with {isInitial = true}
	create Transition
	with {src=s1,tar=s0,symbol=**one** Symbol}

Mutations that produce a non-deterministic automaton	
Create λ-transition	**create** Transition

Create transition with same symbol from a state to a different one	t = **select one** Transition **where** {symbol<>null}
	create Transition
	with { src = t.src, symbol = t.symbol,
	tar = **one** State **where** {**self** <> t.tar}}

Table 1: Using Wodel to define automata mutations

Listing 4 shows part of the code implementing mutation *Create transition* (cf. Table 1). This excerpt creates a transition (lines 3–4), obtains an automaton object from the seed model (lines 8–10), adds the transition to the automaton (lines 12–13), selects a state (lines 15–17), and sets the state as source of the transition (lines 18–19). The listing omits the code for tasks like model loading or checking conformance of the result. Altogether, the mutation amounts to 103 lines of code, empty lines and comments excluded.

```
 1 ...
 2 // create transition
 3 EClass transitionClass = (EClass)epackage.getEClassifier("Transition");
 4 EObject transition = EcoreUtil.create(transitionClass);
 5
 6 // search object automaton in model
 7 EObject automaton = null;
 8 for (TreeIterator<EObject> it = seed.getAllContents(); it.hasNext(); ) {
 9   automaton = it.next();
10   if (automaton.eClass().getName().equals("Automaton")) {
11     // add transition to automaton
12     EStructuralFeature feature = automaton.eClass().
            getEStructuralFeature("transitions");
13     ((List<EObject>)automaton.eGet(feature)).add(transition);
14     // set random state as source of the transition
15     feature = automaton.eClass().getEStructuralFeature("states");
16     List<EObject> states = (List<EObject>)automaton.eGet(feature);
17     EObject randomState = states.get(rand.nextInt(states.size()));
18     feature = transitionClass.getEStructuralFeature("src");
19     transition.eSet(feature, randomState);
20 ...
```

Listing 4: Java code for mutation *Create transition*

4. TOOL SUPPORT

We have built a development environment for WODEL, available as an Eclipse plugin, to mutate EMF models. Fig. 5 shows its architecture. The environment provides an editor for WODEL, built with Xtext[1], which incorporates a validator and code completion facilities to help users in selecting valid class, reference and attribute names from the domain meta-model. Correct WODEL programs are automatically compiled into Java using an Xtend[2] code generator. The produced Java code, which is in charge of creating the mutants from the seed models, can be transparently executed from the WODEL IDE. The advantage of explicitly generating Java code is that it can be used in stand-alone applications. Moreover, this code is generic as it manipulates models reflectively, and hence, it can be reused to mutate any model conformant to the domain meta-model (see Listing 4 for an example of use of the EMF reflective API). In addition, WODEL defines an extension point which allows users to register domain-specific post-processors to be executed upon mutant generation (see Section 5 for an example).

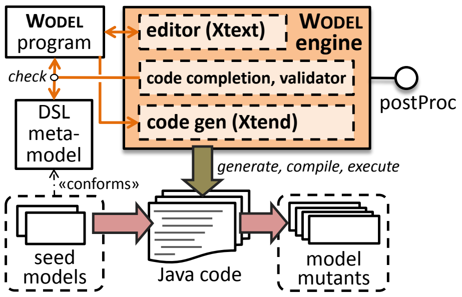
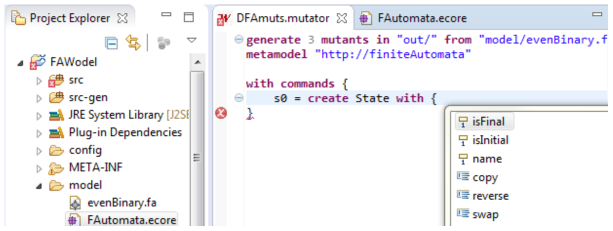


Figure 5: Architecture of Wodel's environment

Fig. 6 shows a screenshot of the IDE illustrating the code completion facilities. In this case, it suggests valid attributes for class State, and some applicable modification operators.



```
generate 3 mutants in "out/" from "model/evenBinary.f
metamodel "http://finiteAutomata"

with commands {
    s0 = create State with {
}
```

Figure 6: Screenshot of the Wodel IDE

5. WODEL-EDU: MODEL MUTATION FOR THE GENERATION OF EXERCISES

For our running example, we have a WODEL program that produces incorrect solutions (i.e., model mutants) from a correct one. In addition, we have built a post-processor which, out from the correct solution and its mutants, generates exercises that can be automatically graded for self-evaluation. Fig. 7 shows its architecture. The post-processor can be configured with a description of the exercises and how model elements should be rendered. This allows generating exercises not only for automata, but for different domains.

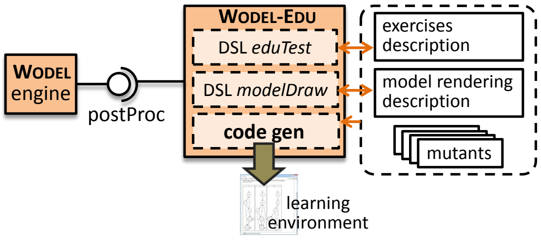


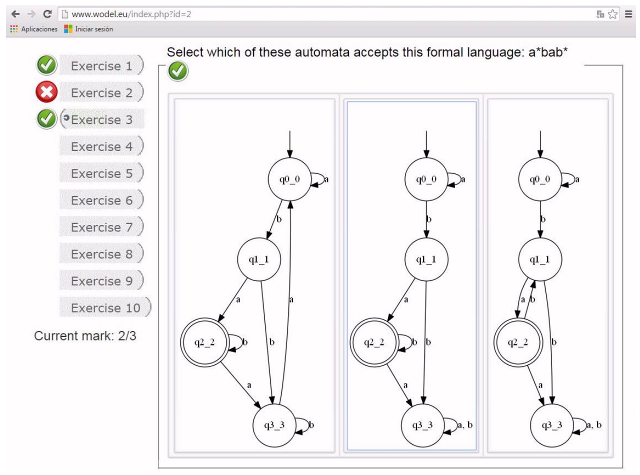
Figure 7: Architecture of the Wodel-Edu plugin

Listing 5 shows a fragment with the description of some exercises. Line 1 states that failed exercises cannot be retried, and that exercises will show both correct and incorrect solutions and students should select the correct one. Instead, selecting showall=no produces exercises where a single model is presented, and students must identify if it is correct.

```
1 retry=no, showall=yes
2 description for 'aut0.fa' = "Select which of these automata accepts
3                              this formal language: a∗bab∗"
4 description for 'aut1.fa' = "Select the automata accepting..."
```

Listing 5: Describing the exercises

Model rendering can be configured using a language similar to the *dot* notation provided by *Graphviz*[3], which is the technology we use to visualize models. Fig. 8 shows a screenshot of the generated application, which contains some automata exercises. The shown exercise consists in selecting the automaton accepting the language a^*bab^*. The automaton in the middle is correct. The other two, which are incorrect, were generated by applying the mutation **modify target** tar **from one** Transition **to other** State, which redirects a random transition to a different target state. This application can be accessed on-line at http://www.wodel.eu.



Select which of these automata accepts this formal language: a*bab*

Current mark: 2/3

Figure 8: Generated application

[1]http://www.eclipse.org/Xtext/
[2]http://www.eclipse.org/xtend/
[3]http://www.graphviz.org/

WODEL-EDU can be used to create similar exercises for other domains (e.g., class diagrams) by providing a WODEL program with the mutations of interest, how model elements are visualized, and a description of the exercises as in Listing 5.

6. RELATED RESEARCH

Mutation is used in areas like model-based testing, program testing [7], evaluation of clone-detection algorithms [11], generation of large model sets [8], education [9], or evolutionary algorithms [6]. While most of these systems are built ad-hoc, WODEL may automate their construction.

The mutation framework in [1] is specific to model transformation testing, and mutation operators are defined with the Kermeta model management language[4]. Instead, WODEL has primitives tailored to mutation, and is not restricted to mutation testing. Also for mutation testing, MuDeL [4] allows describing mutation operators for grammar-based artefacts, typically programs. MuDeL is based on replacement patterns, and WODEL on operations. Moreover, WODEL has further facilities to combine mutation operators and discard malformed or duplicated mutants. In the area of mutation-based testing, Muta-Pro [12] uses techniques to detect equivalent mutants, whereas we use model comparison.

Model-based mutation testing has been applied to adaptive systems [2], model-based delegation security policies [7] and logic formula [5]. It has also been used with Simulink models to compare clone-detection algorithms [11]. In these cases, mutation operators were manually encoded using low-level languages, and the frameworks were built ad-hoc. Tools like WODEL may help to improve development automation.

The SiDiff framework [8] allows creating large models with some statistical properties. While the authors only illustrate creation operations, the creation context can be selected via stochastic properties. Other tools to generate sets of large models include the Ecore Mutator[5], which provides a programmatic API to code mutations in plain Java. WODEL can be used for model generation as well, but it is more general as it provides primitives for deletion and modification.

Similar to our application WODEL-EDU, in [9], mutation is used to generate exercises for state machines for a massive open online course. However, [9] is work in progress and the authors aim at building the system by hand. This could be done automatically with WODEL. Moreover, WODEL-EDU can be applied to generating exercises in any domain.

In summary, our proposal is novel as current mutation-based systems are commonly built by hand. The few existing languages to define mutations [1] focus on testing and work over grammars. As many applications need to specify and produce mutants, an extensible approach like ours is useful.

7. CONCLUSIONS AND FUTURE WORK

This paper has presented WODEL, a DSL to specify domain-specific mutation operators and mutation programs. WODEL is domain-independent, and its development environment can be extended for different applications. In this work, we showed an application in the education domain.

[4]http://www.kermeta.org/documents/
[5]https://code.google.com/a/eclipselabs.org/p/ecore-mutator/

In the future, we will enrich the languages of the WODEL-EDU plugin to support more complex learning environments (e.g., including gamification) and exercises (e.g., interactive exercises where students have to correct an incorrect solution). We also plan to develop plugins for WODEL for other areas, like model-based testing and evolutionary computation, which might trigger improvements in WODEL itself.

Acknowledgements. Thanks to Víctor López Rivero for the initial implementation of the Wodel IDE. Work supported by the Spanish Ministry of Economy and Competitivity (TIN2014-52129-R), the Madrid Region (S2013/ICE-3006), and the EU commission (FP7-ICT-2013-10, #611125).

8. REFERENCES

[1] ARANEGA, V., MOTTU, J., ETIEN, A., DEGUEULE, T., BAUDRY, B., AND DEKEYSER, J. Towards an automation of the mutation analysis dedicated to model transformation. *STVR 25*, 5-7 (2015), 653–683.

[2] BARTEL, A., BAUDRY, B., MUNOZ, F., KLEIN, J., MOUELHI, T., AND TRAON, Y. L. Model driven mutation applied to adaptative systems testing. In *ICST Workshops* (2011), pp. 408–413.

[3] BRAMBILLA, M., CABOT, J., AND WIMMER, M. *Model-Driven Software Engineering in Practice*. Morgan & Claypool, USA, 2012.

[4] DA SILVA, A., AND MALDONADO, J. C. MuDeL: a language and a system for describing and generating mutants. *J. Braz. Comp. Soc. 8*, 1 (2002), 73–86.

[5] HENARD, C., PAPADAKIS, M., AND TRAON, Y. L. Mutalog: A tool for mutating logic formulas. In *ICST Workshops Proceedings* (2014), IEEE CS, pp. 399–404.

[6] MOAWAD, A., HARTMANN, T., FOUQUET, F., NAIN, G., KLEIN, J., AND BOURCIER, J. Polymer - A model-driven approach for simpler, safer, and evolutive multi-objective optimization development. In *MODELSWARD* (2015), SciTePress, pp. 286–293.

[7] NGUYEN, P. H., PAPADAKIS, M., AND RUBAB, I. Testing delegation policy enforcement via mutation analysis. In *ICST Workshops* (2013), pp. 34–42.

[8] PIETSCH, P., YAZDI, H., AND KELTER, U. Controlled generation of models with defined properties. In *SE* (2012), vol. 198 of *LNI*, GI, pp. 95–106.

[9] SADIGH, D., SESHIA, S. A., AND GUPTA, M. Automating exercise generation: A step towards meeting the MOOC challenge for embedded systems. In *WESE* (2013), ACM, pp. 2:1–2:8.

[10] STEINBERG, D., BUDINSKY, F., PATERNOSTRO, M., AND MERKS, E. *EMF: Eclipse Modeling Framework, 2^{nd} Edition*. Addison-Wesley Professional, 2008.

[11] STEPHAN, M., ALALFI, M. H., STEVENSON, A., AND CORDY, J. R. Using mutation analysis for a model-clone detector comparison framework. In *ICSE* (2013), IEEE / ACM, pp. 1261–1264.

[12] VINCENZI, A. M. R., DA SILVA, A., DELAMARO, M. E., AND MALDONADO, J. C. Muta-Pro: Towards the definition of a mutation testing process. *J. Braz. Comp. Soc. 12*, 2 (2006), 49–61.