A Domain-Specific Language for Model Mutation and its Application to the Automated Generation of Exercises

Pablo Gómez-Abajo, Esther Guerra, Juan de Lara

Modelling & Software Engineering Research Group
http://miso.es
Computer Science Department
Universidad Autónoma de Madrid (Spain)

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 the application of 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 conformant to arbitrary meta-models. Its development environment is extensible, permitting the incorporation of post-processors for different applications. In particular, we describe Wodel-Edu, a post-processing extension directed to the automated generation of exercises for particular domains and their automated correction. We show the application of Wodel-Edu to the generation of exercises for deterministic automata, and report on an evaluation of the quality of the generated exercises, obtaining overall good results.

Keywords: Model-Driven Engineering, Domain-Specific Languages, Model Mutation, Education, Automatic Exercise Generation and Correction

Email addresses: `Pablo.GomezA@uam.es` (Pablo Gómez-Abajo), `Esther.Guerra@uam.es` (Esther Guerra), `Juan.deLara@uam.es` (Juan de Lara)

1. Introduction

Model-Driven Engineering (MDE) [1, 2] uses models in all phases of the software development process to specify, simulate, test and generate code for the final system, among other activities. Such models are usually defined using Domain-Specific Languages (DSLs) specialized for the particular application domain. Domains where MDE and DSLs have been applied successfully include concurrent programming [3], reactive process control programming [4], business logic modelling [5], web modelling [6], and smart card software development [7], just to name a few.

Since models are the primary asset in MDE, model manipulations become a key activity in this paradigm. For this purpose, DSLs particularly tailored to the model transformation task are heavily used. For example, many DSLs exist to specify model simulators, to produce a model from another one, to migrate models [8], or to synthesize code [9].

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 [10], a transformation is represented as a model that is mutated to evaluate the efficacy of a test 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 (e.g., consisting in the identification of the errors injected by the mutations) that can be automatically graded [11].

There are some frameworks for model mutation, but they are specific for a language (e.g., logic formulae [12]) or domain (e.g., testing [10, 13]); moreover, mutation operators are normally created using general-purpose programming languages that are 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. The language 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 using our tool Wodel-Edu. The tool has been developed as an Eclipse plugin and contributed as a post-processor of Wodel. Wodel-Edu is able to generate three types of exercises, combining several DSLs to describe the exercises, the visual rendering of the models, and the textual description of the applied mutations. We have conducted a user evaluation of the perceived quality of the generated exercises, which shows overall good results.

This is a revised version of our previous paper [14] with the following novel contributions. We have extended our DSL Wodel with the possibility to define blocks which use the mutants generated by other blocks as seed models, and we enable traceability support by maintaining a registry of applied mutations that can be compacted to eliminate mutations that cancel each other (e.g., a mutation creates an object, and another deletes the object). To demonstrate the applicability of Wodel, we use it to mutate programs for the purpose of mutation-based testing. We also explain in detail our application Wodel-Edu for the generation of exercises, showing three new DSLs that help in fine-tuning the generation of exercises. Wodel-Edu itself has been extended with a new kind of exercise that benefits from the registry of applied mutations. Moreover, we present the results of a user study aimed at evaluating the quality of the automatically generated exercises. Finally, we have improved the analysis of related works.

The remainder of this paper is organised as follows. First, Section 2 gives an overview of our approach and introduces a running example. Next, Section 3 presents the Wodel language, and Section 4 its tooling. Section 5 applies Wodel to the generation of test exercises by the mutation of models. The section also includes a user study to evaluate the quality of the exercises so generated. Section 6 discusses related works, and finally, Section 7 concludes the paper and identifies some lines of future work.

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 arbitrary meta-models, and to facilitate the use of the generated mutants for different applications.

Figure 1 shows the workflow of our approach. First, the user provides a set of seed models conformant to a meta-model (label 1). Then, the user defines the desired mutation operators and their execution details using Wodel, 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.

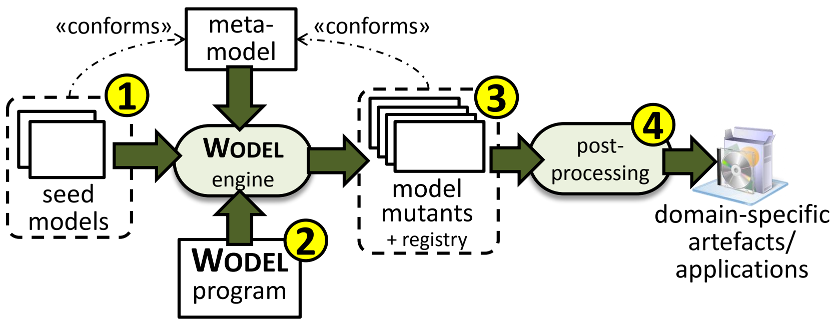


Figure 1: Scheme of our approach

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. Moreover, for traceability reasons, it is possible to produce a registry of the mutations applied to each mutant together with information of their application context. 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

Along the paper, we use Wodel for the mutation of automata and illustrate our proposal with an application of model mutation to education. In particular, we will generate three kinds of exercises by using model mutation. In the first kind, 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. In the second kind, students are presented an automaton, and they have to decide whether it is correct or not. The presented automaton is selected randomly among the seed model and its mutants. The third kind of exercise shows an incorrect automaton and a list of possible actions over the automaton (e.g., reversing a transition or make a state final), and students have to select the actions that would fix the automaton. In this case, the text of the correct actions is synthesized by reversing the mutations used to generate the incorrect automaton from the correct one. For this purpose, we make use of our registry of mutations. Our approach permits generating exercises with different degrees of difficulty and supports automatic correction.

Figure 2 shows the meta-model for automata used in the running 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 an attached symbol; if it lacks a symbol, it is considered a λ-transition. The meta-model includes three OCL invariants that any Automaton must fulfil: the first one demands the existence of exactly one initial state, the second one demands at least one final state, and the last requires distinct symbols in the alphabet.

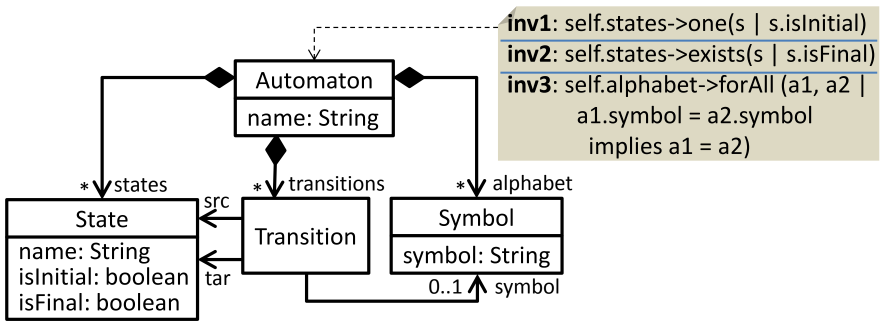


Figure 2: Meta-model for finite automata

3. Wodel: A Domain-Specific Language for Model Mutation

In this section, we first introduce our DSL Wodel and illustrate its usage showing examples of mutation operators for finite automata conformant to the meta-model in Figure 2. Then, in Section 3.2, we explain our mechanism to provide traceability between the seed model and its mutants via a registry of applied mutations. In Section 3.3, we present an analytical evaluation of the expressiveness and succinctness of Wodel. Finally, to show the applicability of Wodel, Section 3.4 presents its application to the mutation of a simple programming language for the purpose of mutation-based testing.

3.1. The Wodel *language*

Wodel programs have two parts. The first one declares the number of mutants to generate, the output folder, the seed models and their meta-model. The second part defines mutation operators and how many times they should be applied. Optionally, programs can also include a list of OCL constraints that all generated mutants should fulfil.

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 makes it 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 modified 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. Figure 3 shows an excerpt of the Wodel meta-model with the definition of some representative mutation primitives. All mutation kinds inherit from class 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:

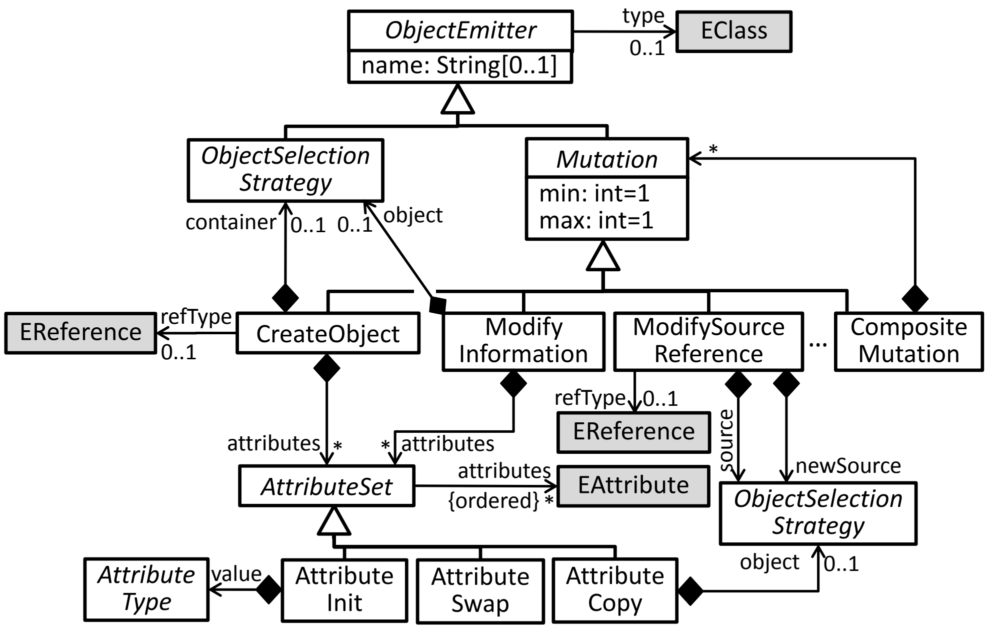


Figure 3: Some supported mutations

- 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** 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 or attribute, WODEL assigns it one object or value 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 [15] 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 when writing WODEL programs.

- CreateReference: It creates a new reference of the given type between two objects. The objects may be selected using an ObjectSelectionStrategy. If no object is selected, then 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 one. 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: They redirect 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 obtained 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 (e.g., the value of some of their features), so that they can be used in subsequent mutations.

Object and references used in mutations and selectors can be chosen 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 Figure 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 that is instance of 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**}).

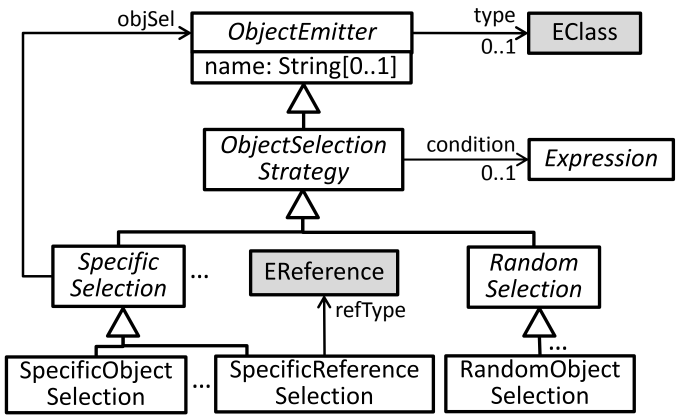


Figure 4: Some supported selection strategies

As Listing 1 shows, WODEL has a textual concrete syntax. A graphical syntax – perhaps similar to graph transformation rules [16] – could have been chosen instead to express the mutation operators. However, we opted for a textual one due to its conciseness, the need to include expressions for attribute mutation, to facilitate referencing mutations from other ones, and to be able to aggregate them into composite mutations. Exploring visual notations to represent parts of a WODEL program (e.g., the block dependencies in Listing 4) is left for future work.

Listing 2 shows a brief excerpt of WODEL's grammar. The first part of a WODEL program (DEFINITION) declares the meta-model to be used, the number of mutants to be generated, and where to find the seed models. The definition is followed by a sequence of mutation commands (MUTATION*), either executed sequentially or embedded in blocks (BLOCK*).

```
WODELPROGRAM ::= DEFINITION
  with ( commands { MUTATION∗ } | blocks { BLOCK∗ } )
  ( constraints { CONSTRAINT∗ } )?

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 the WODEL grammar

Listing 3 shows another example. It generates 3 mutants from every seed model in folder models (line 1). Each mutant is obtained by applying the specified mutations a random number of times according to the mutation cardinality interval. Lines 5–7 define a composite mutation which removes a random non-initial State (line 5), as well as 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 an arbitrary Transition, and modifies its reference symbol to point to a different Symbol. Both mutations declare a cardinality interval controlling the number of applications in every mutant: the first mutation will be applied a random number of times with uniform probability between 0 and 2, and the second one between 1 and 3.

```
1 generate 3 mutants in "out/" from "models/"
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
```

```
          to other Symbol [1..3]
}
```

Listing 3: Composite mutation and cardinalities

In Listings 1 and 3, all mutations are executed in sequence on the seed models specified in the program header. In addition, WODEL also permits organising mutations in named blocks, which take as seed either the mutants generated by some selected previous blocks, or the seed models given in the program header if no other block is indicated. Listing 4 shows an example. It declares two blocks, one called first, which generates 2 mutants from the default seed model evenBinary.fa (lines 5–7), and another called second, which generates 3 mutants from each mutant produced by block first (lines 8–10). The directive repeat=no in the second block ensures that it will generate mutants different from the used seed models (i.e., different from the mutants produced by block first).

```
 1 generate mutants in "out/" from "evenBinary.fa"
 2 metamodel "http://fa.com"
 3
 4 with blocks {
 5   first {
 6     remove one Transition
 7   } [2]
 8   second from first repeat=no {
 9     create Transition
10   } [3]
11 }
```

Listing 4: Mutation blocks

Implementation-wise, our WODEL programs handle each defined mutation as an operation, and therefore, it is not possible to have contradictory mutations. For example, if a mutation operation deletes an object, then a subsequent mutation operation cannot select the deleted object, e.g., to modify its attributes. However, two mutations may cancel each other (e.g., one mutation creates an object, and another deletes it).

3.2. Traceability in WODEL

Sometimes, mutation-based applications require having traceability between the seed models and the generated mutants, or having knowledge of the operations used to produce the mutants. For instance, in Section 5, we need to access the list of mutations used to generate an incorrect automaton in order to be able to synthesize sentences that explain how to correct it.

Thus, when a WODEL program is executed, it is possible to generate a registry with the sequence of applied mutations and their application context for each generated mutant. Every registry is itself a model, which facilitates being queried or manipulated by any post-processing application making use of WODEL.

Figure 5 shows an excerpt of the mutations registry meta-model. The Registry contains an ordered list of AppliedMutation objects, each one of them storing a reference to the executed Mutation command in the WODEL program used to generate the mutant. This is possible because WODEL programs are models as well, conformant to the WODEL meta-model shown in Figures 3 and 4. Depending on the particular command executed, the appropriate subclass of AppliedMutation is instantiated. For example, for each execution of a CreateObject command in a program, an ObjectCreated object is added to the registry model storing a reference to the command and to the created object. While some of the records in the registry need to refer to elements in the seed model (e.g., the objects removed from the seed model are not present in the mutant), others need to refer to the resulting mutant (e.g., the objects created as a result of a mutation do not belong to the seed model).

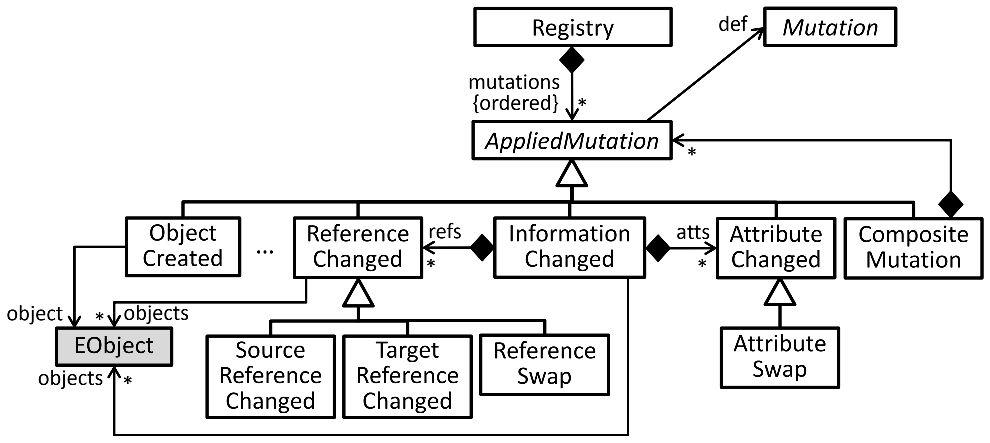


Figure 5: Meta-model of the registry of applied mutations (excerpt)

The registry records *all* applied mutations. This means that even if the effect of a mutation application is undone by a subsequent mutation (e.g., the value of a boolean attribute is reversed twice), both mutations are kept in the registry. Since some applications may consider irrelevant the mutations which get finally undone, we have extended WODEL with a post-processor step which removes from the registry the mutations that cancel each other, thus obtaining a more compact description of the differences between the seed model and the generated mutant. For example, if the registry records the creation of an object and its subsequent deletion, then both applied mutations are removed from the registry, as well as any intermediate modification of the object attributes. Compacting the registry in this way is optional.

3.3. Evaluation of the expressiveness 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 [11]. Note that [11] does not consider final states or mutate transition symbols as we do, but it applies different number of mutations which we can 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 for model elements. 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. A more thorough analysis remains as future work.

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 [15], 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 with respect to the meta-model, model serialization, checking well-formedness of resulting mutants, comparing for equal resulting mutants, or producing a registry with the applied mutations.

To illustrate the complexity of the equivalent Java code, Listing 5 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

Mutations that change the language

Create transition [11]	**create** Transition **with** {symbol = **one** Symbol}
Create final state	**create** State **with** {isFinal = true}

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	**remove one** State **where** {isInitial = false}
	remove all Transition **where** {src = null}
	remove all Transition **where** {tar = null}

Change symbol in transition	**modify target** symbol **from one** Transition **to other** Symbol

Change final state to non-final	**modify one** State **where** {isFinal = true}
	with {**reverse**(isFinal)}

Change initial state to a different one [11]	s0 = **modify one** State **where** {isInitial = true}
	with {isInitial = false}
	s1 = **modify one** State **where** {**self** <> s0}
	with {isInitial = true}

Swap direction of transition [11]	**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 [11]	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 mutations for automata

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 and meta-model loading or checking the conformance of the result. Altogether, the mutation amounts to 103 lines of code, empty lines and comments excluded. Instead, the same functionality is obtained using 1 line in Wodel.

```
1 ...
2 // create transition
```

```
EClass transitionClass = (EClass)epackage.getEClassifier("Transition");
EObject transition = EcoreUtil.create(transitionClass);

// search object automaton in model
EObject automaton = null;
for (TreeIterator<EObject> it = seed.getAllContents(); it.hasNext(); ) {
  automaton = it.next();
  if (automaton.eClass().getName().equals("Automaton")) {
    // add transition to automaton
    EStructuralFeature feature = automaton.eClass().getEStructuralFeature("transitions");
    ((List<EObject>)automaton.eGet(feature)).add(transition);
    // set random state as source of the transition
    feature = automaton.eClass().getEStructuralFeature("states");
    List<EObject> states = (List<EObject>)automaton.eGet(feature);
    EObject randomState = states.get(rand.nextInt(states.size()));
    feature = transitionClass.getEStructuralFeature("src");
    transition.eSet(feature, randomState);
...
```

Listing 5: Java code for mutation *Create transition*

3.4. Applicability to program mutation

To demonstrate the applicability of Wodel beyond our running example, we show how to use it to perform program mutation with the goal of mutation-based testing [17]. This testing technique permits evaluating the quality of a set of test cases. For this purpose, the program under test is mutated to inject faults, and the test cases are executed on the created program mutants. If the set of test cases does not detect the injected faults, then the set is insufficient and should be extended to catch those errors.

We will illustrate program mutation for an adapted version of Asple [18], which is a programming language derived from Algol 68. Asple provides the basic features of general-purpose programming languages: variable declaration, assignments, conditionals, loops, read input/write output data, expressions and literals. Figure 6 shows an excerpt of the meta-model we have built for this programming language.

The left of Figure 7 shows an Asple program example. The program declares and initializes two variables (lines 2–4), and executes a loop that increases the value of variable n until its value reaches 10 (lines 5–7). To the right, we show part of the abstract syntax of this program, omitting the variable initializations in lines 3–4 for simplicity.

Table 2 gathers the program mutations that we have implemented for Asple, and which were proposed in [19, 20]. As Wodel is meta-model independent, these mutations can be adapted easily to models of other general-purpose programming languages. The first two mutations replace an existing

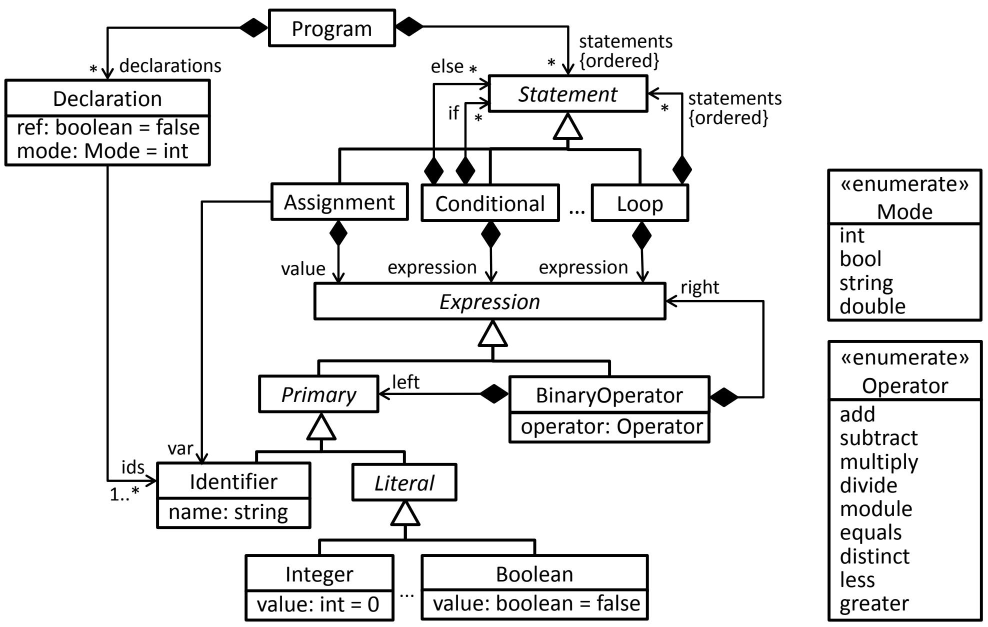


Figure 6: Meta-model of the ASPLE programming language (excerpt)

```
begin
  int n, inc;
  n = 0;    // omitted
  inc = 1; // omitted
  while (n < 10) {
     n = n + inc;
  }
end
```

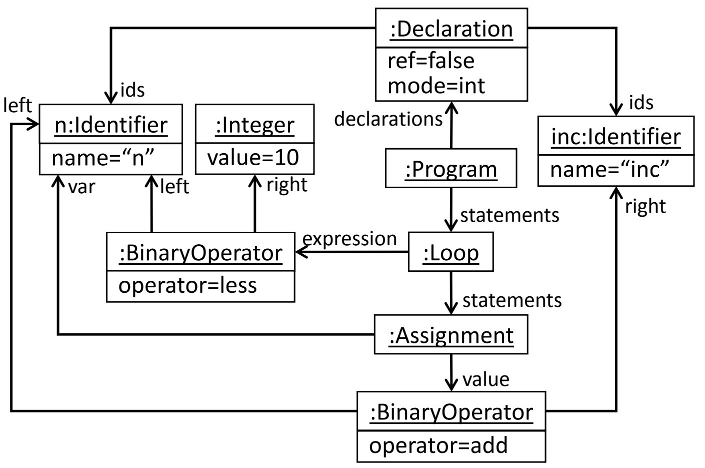


Figure 7: A simple ASPLE program. Textual representation (left) and excerpt of abstract syntax (right)

arithmetic or relational operator by a different one. Both mutations use the `in` operator, which selects a random element in the given set, and when used within a `modify ... with` command, ensures that the selected element is different from the old attribute value. The third mutation selects an arbitrary Loop statement and changes its end condition (i.e., expression) to true; the old loop expression gets automatically deleted. The fourth mutation deletes a random statement from the program; in case of conditionals and loops, their inner statements get automatically deleted as well. The fifth mutation modifies an integer literal with a different, random value. Finally, the last mutation randomly selects any arithmetic expression and increments its value by 1.

Replace an arithmetic operator by another one [20]	**modify one** BinaryOperator **where** {operator **in** [add, subtract, multiply, divide, module]} **with** {operator **in** [add, subtract, multiply, divide, module]}
Replace relational operator by another one [20]	**modify one** BinaryOperator **where** {operator **in** [less, equals, greater, distinct]} **with** {operator **in** [less, equals, greater, distinct]}

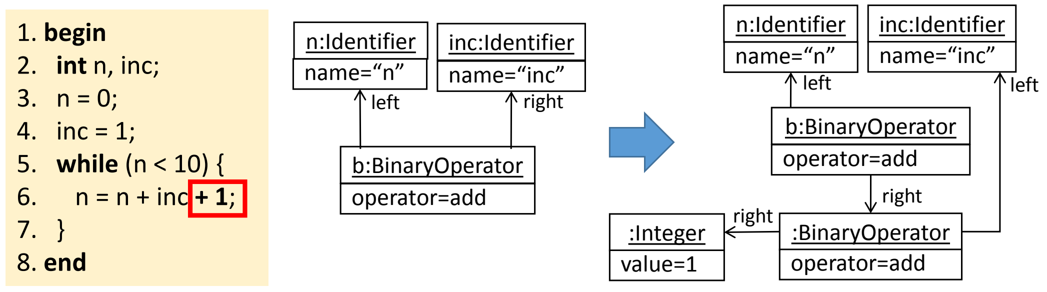
Replace loop end condition by infinite loop [20]	loop = **select one** Loop
	create Boolean **in** loop.expression **with** {value = true}

Delete statement [20]	**remove one** Statement
Replace scalar variable [20]	**modify one** Integer **with** {value = **random**}

Increment an arithmetic expression by 1 [19]	op = **select one** BinaryOperator
	      where {operator **in** [add,subtract,multiply,divide,module]}
	exp= **create** BinaryOperator **in** op.right
	      with {left = op.right, operator = add}
	create Integer **in** exp.right **with** {value = 1}

Table 2: Using Wodel to define program mutations

As an example, the left of Figure 8 shows the result of applying the last mutation in Table 2 to the program in Figure 7. The mutation selects the operation in line 6, and increments its result by 1. To the right, the figure shows the elements affected by the mutation before and after it takes place, using an abstract syntax representation.



```
1. begin
2.   int n, inc;
3.   n = 0;
4.   inc = 1;
5.   while (n < 10) {
6.     n = n + inc + 1;
7.   }
8. end
```

Figure 8: Mutant of the Asple program in Figure 7

This example illustrates how Wodel can be applied to different meta-models even if the underlying DSL is textual – as in the case of Asple – and can be used for injecting faults in programs using mutations. A full development of a mutation-based framework based on Wodel is left for future work. In this paper, we focus on the application of Wodel to the educational domain.

4. Tool Support

We have built a development environment for Wodel, available as an Eclipse plugin [21], to mutate EMF models. The tool is freely available at `http://miso.es/tools/Wodel.html` including the source code, installation instructions, and videos.

Figure 9 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.

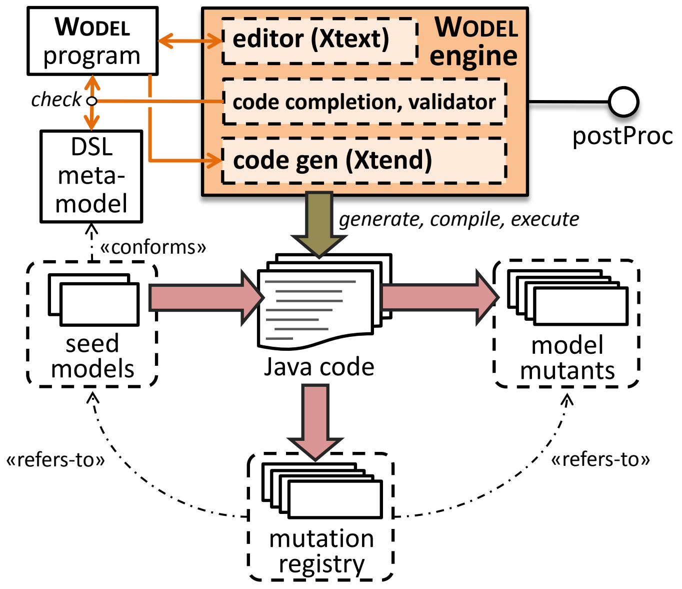


Figure 9: Architecture of Wodel's environment

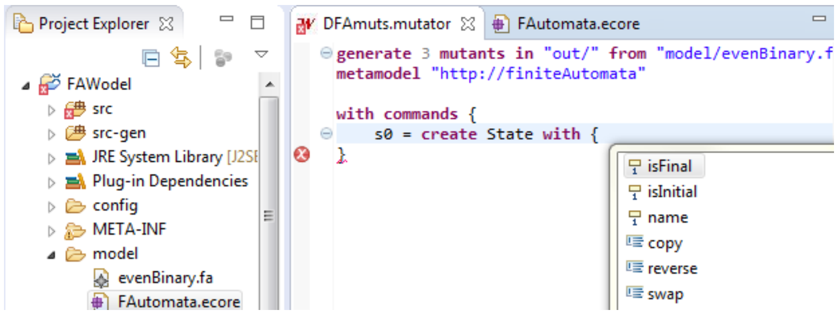
Correct Wodel programs are automatically compiled into Java code us-

[1]`http://www.eclipse.org/Xtext/`

ing an Xtend[2] code generator. The produced Java code, which is in charge of creating the mutants from the seed models as well as the registry of applied mutations, can be transparently executed from the WODEL IDE. The advantage of explicitly generating Java code is that it can be used in standalone 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 5 for an example of use of the EMF reflective API).

In addition, WODEL defines an extension point (postProc in the figure) which allows users to register domain-specific post-processors to be executed upon mutant generation. In this paper, we report on two particular instantiations of this extension point, namely, a facility to compact the generated mutation registry (see Section 3.2), and an application for the automatic generation of exercises (see Section 5).

Figure 10 shows a screenshot of the IDE illustrating the code completion facilities. In this case, the IDE suggests the name of valid attributes for class State, and some applicable modification mutation operators.



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

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

Figure 10: Screenshot of the WODEL IDE

5. Wodel-Edu: Model Mutation for the Generation of Exercises

In this section, we present an application of WODEL to the generation of exercises. In particular, we have built a post-processor called WODEL-EDU which, starting from a WODEL program that produces incorrect solutions

[2]http://www.eclipse.org/xtend/

(i.e., model mutants) from a correct seed model, it generates a web application with exercises that can be automatically graded for self-evaluation. We support the generation of three kinds of exercises of increasing complexity (see the different generation schemes in Figure 11):

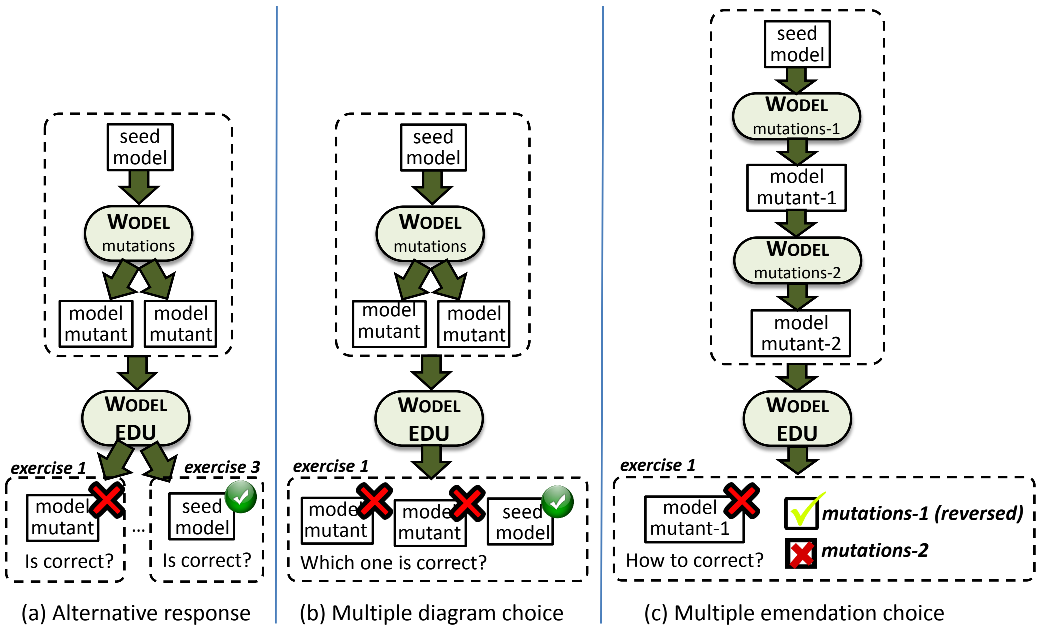


Figure 11: Generation scheme of the three types of exercise

- *Alternative response*: These exercises show a diagram, and students must decide whether it is correct or not. This is the simplest kind of exercise, where the diagram is correct in case it corresponds to the seed model, or incorrect if it corresponds to one of the mutants (see Figure 11a).
- *Multiple diagram choice*: These exercises show several diagrams among which only one is correct, and students must identify it. Hence, this kind of exercise just has to present the diagram corresponding to the seed model without mutations among its mutated versions (see Figure 11b).
- *Multiple emendation choice*: This is the most complex kind of exercise. In this case, the exercise shows an incorrect diagram generated

by applying one or several mutations on a seed model, as well as the description of several possible emendations over the diagram, and students have to choose the subset of emendations that would fix the diagram. The correct emendations are automatically synthesized by reversing the mutations applied to generate the incorrect diagram. The incorrect emendation options are generated by mutating the already incorrect diagram in order to obtain modification actions that make sense for the exercise at hand (see Figure 11c).

As it can be observed, there are different aspects of the exercise generation that need to be configured. First, the kind of exercises to be generated and the text description of each exercise. Second, how the models are being rendered graphically, as the model representation will depend on the exercises domain (e.g., automata, class diagrams, etc.). Finally, for the exercises of type *multiple emendation choice*, the text of the different emendation options is generated from the applied mutations, and therefore, it may be necessary to fine-tune this text. Our approach to describe these aspects is via a family of cooperating DSLs.

In the following subsections, we present the architecture of Wodel-Edu and how it can be configured for the generation of exercises in different domains using DSLs (Section 5.1). Then, we exemplify the generation of the different kinds of exercises in the finite automata domain (Section 5.2). Finally, we present the results of a user study aimed at measuring some aspects of the quality of the exercises automatically generated with Wodel-Edu (Section 5.3).

5.1. A family of DSLs for exercise generation

Figure 12 shows the architecture of our Wodel-Edu post-processor. It provides four DSLs to configure the text and style of exercises (eduTest), how model elements should be graphically rendered (modelDraw), how to represent a model element textually (modelText), and how to represent an applied mutation textually (mutaText). These DSLs facilitate the customisation of the generated exercises for different domains (e.g., automata, class diagrams, electronic circuits, etc.). While eduTest and modelDraw must be used always to configure any kind of exercise, modelText and mutaText are only used in exercises of type *multiple emendation choice* when there is the need to override the text of the emendations that Wodel-Edu synthesizes by default. In the following, we illustrate the four DSLs for the generation of finite automata exercises.

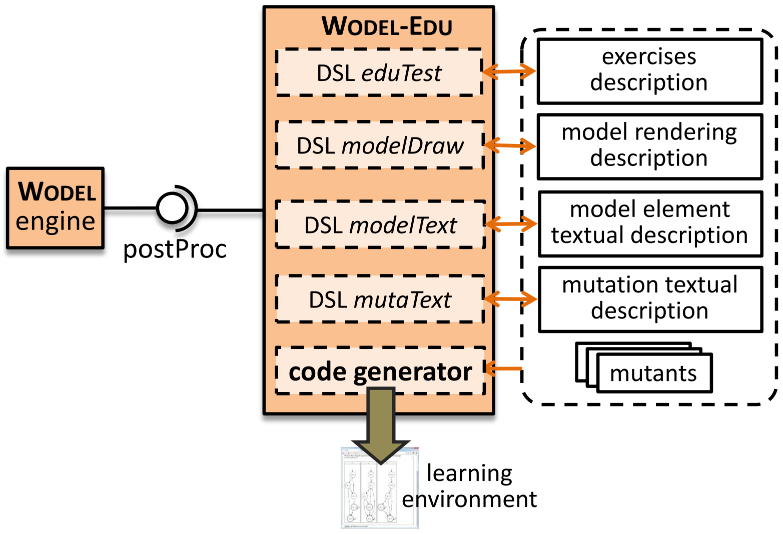


Figure 12: Architecture of the Wodel-Edu plugin

5.1.1. *Describing exercises with* eduTest

Listing 6 shows a fragment with the description of some exercises using the eduTest DSL. Line 1 states that the generated exercises can be resolved in any order, and it is possible to navigate back and forward to new or already answered exercises. Lines 3–8 define two *multiple choice emendation* exercises with the following characteristics: retry is allowed in case of failure (retry=yes); all exercises have the same weight in the final grade (weighted=no); there is no penalty in case of failure (penalty=0.0); exercises are shown in descending order of their number of emendation options (order=options-descending, alternatively, they could be shown in ascending order, in the definition order, or randomly); all emendations needed to fix a diagram are grouped as a single option (mode=radiobutton, alternatively, each necessary emendation could be shown in a separate checkbox, which may require selecting one or more of them to solve the exercise correctly). The text of the exercises, as well as their seed model, are declared in lines 6–7. Below, lines 10–13 define one exercise of type *multiple choice diagram*.

```
1  navigation=free
2
3  MultiChoiceEmendation complex {
4    retry=yes, weighted=no, penalty=0.0,
5    order=options−descending, mode=radiobutton
6    description for 'exercise4.model' = 'Select the required changes so that the automaton accepts a+b+'
7    description for 'exercise6.model' = 'Select the required changes so that the automaton accepts a∗b'
}

MultiChoiceDiagram simple {
  retry=no
  description for 'exercise1.model' = 'Select which of these automata accepts the language a∗bab∗'
}
```

Listing 6: Defining the exercises with eduTest

5.1.2. Describing model visualization with modelDraw

Model rendering is configured using the DSL modelDraw. This is a simple language similar to the *dot* notation provided by *Graphviz*[3], which is the technology used by Wodel-Edu to visualize models. As an example, Listing 7 shows the use of the DSL to describe the graphical appearance of finite automata. The first line specifies the domain meta-model, which enables content assistance and type-checking. Then, for each class in the meta-model, it is possible to configure whether it will be shown as some kind of node (circle, double circle, ellipse, rectangle, etc.) or as an edge. Nodes and edges can display a label with the value of some attribute of the corresponding class. By convention, if no label is provided, we show the content of the attribute name (if it exists). Finally, we also support assigning different visualizations to the same class depending on the value of some boolean attribute. In our case, the listing declares that initial states will be shown as ticked nodes (line 4), non-final states as circles (line 5), final states as double circles (line 6), and transitions as edges labelled with the transition symbol (line 7).

```
1 metamodel "http://fa.com"
2
3 Automaton: diagram {
4   State (isInitial): markednode
5   State (not isFinal): node, shape=circle
6   State (isFinal): node, shape=doublecircle
7   Transition (src, tar): edge, label=symbol
8 }
```

Listing 7: Defining the graphical rendering of models with modelDraw

5.1.3. Textual description of mutations with modelText *and* mutaText

By default, when Wodel-Edu has to generate a textual description of an object, the object gets described by the value of its attribute name if it

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

exists, or by the name of its class otherwise. The DSL modelText permits overriding this default text for selected meta-model types. In this way, for each class and relation, we can specify a text template where the expressions preceded by the symbol `%` are evaluated on the object and their value is emitted in the resulting string.

Listing 8 shows the usage of the DSL for the running example. According to line 3, whenever an object of type `State` is to be mentioned in some text, it will be written as "State" followed by the name of the state (e.g., "State `q0`" if the state is named "q0"). Similarly, line 4 defines that `Transition` objects will be represented by the text "Transition" followed by the symbol attached to the transition. This symbol is obtained by navigating through the `symbol` reference of the `Transition` object, and then accessing attribute `symbol` (see the meta-model for automata in Figure 2). Finally, in lines 5–6, references `src` and `tar` of `Transition` objects are configured to be written as "source" and "target".

```
1 metamodel "http://fa.com"
2
3 > State: State %name
4 > Transition: Transition %symbol.symbol
5 > Transition.src: source
6 > Transition.tar: target
```

Listing 8: Defining the textual description of model elements with modelText

Similarly, the DSL mutaText permits overriding the default text that Wodel-Edu generates to represent each applied mutation. This text is the one shown as emendation options in the third kind of exercises.

Listing 9 shows an example of mutaText. It defines the alternative text for mutations `TargetReferenceChanged` and `AttributeChanged`, the latter only applicable in case of mutating the value of an attribute of `State`. The DSL allows configuring the specific text to show when the emendation option is correct (lines 4 and 8) and when it is not (lines 5 and 9). Moreover, it permits the use of some predefined variables that contain information about the applied mutation, like `%object` which identifies the mutated object, or `%refName` which contains the name of the reference used in the mutation. These variables will return the textual representation of the object or reference specified with modelText, or a default textual representation if none was given. For instance, the text defined in line 4 of Listing 9, combined with the use of the modelText definition in Listing 8, will generate emendations like this one: `Change Transition a from State s0 to State s1 with new target State q2`. Should we did not use the modelText definition, we would obtain the following emendation instead: Change Transition from s0 to s1 with new tar q2.

```
metamodel "http://fa.com"

> TargetReferenceChanged:
  Change %object from %fromObject to %toObject with new %refName %oldToObject /
  Change %object from %fromObject to %oldToObject with new %refName %toObject

> AttributeChanged (State):
  Change attribute %attName from %object with value %newValue to %oldValue /
  Change attribute %attName from %object with value %oldValue to %newValue
```

Listing 9: Defining the textual description of mutations with mutaText

Altogether, starting from the definition of the exercises and the graphical rendering of models, Wodel-Edu generates a web application with one page for each defined set of exercises. Optionally, the DSLs modelText and mutaText allow customising the text of the options shown in the *multiple emendation choice* exercises, or otherwise, a sensible text for the options is generated by default.

5.2. Generation of exercises by model mutation

Next, we illustrate the different kinds of exercises generated by Wodel-Edu. We start by defining the Wodel program in Listing 10, which creates a set of mutants from the seed automaton models in folder model. The program includes an OCL constraint which demands all states in the generated mutants to be reachable from the initial state, in order to avoid the generation of trivially incorrect automata (lines 30–33). From the generated mutants, Wodel-Edu generates a web application with exercises, which can be accessed on-line at `http://www.wodel.eu/comlan16`.

```
1  generate mutants in "out/" from "model/"
2  metamodel "http://fa.com"
3
4  with blocks {
5    // mutations for exercises of type alternative−response
6    alternative {
7      s0 = select one State where {isInitial = true}
8      s1 = select one State where {isFinal = false}
9      t0 = select one Transition where {src = s0}
10     modify one Transition where {tar = s1} with {swapref(tar, t0.tar)}
11   } [1]
12
13   // mutations for exercises of type multiple−diagram−choice
14   multiple {
15     modify target tar from one Transition to other State
16   } [2]
17
18   // mutations for exercises of type multiple−emendation−choice
19   incorrect_automaton {
```

```
20      modify target tar from one Transition to other State
21      modify one State with {reverse(isFinal)}
22    } [3]
23    incorrect_emend from incorrect_automaton repeat=no {
24      modify target tar from one Transition to other State
25      modify one State with {reverse(isFinal)}
26    } [6]
27  }
28
29  constraints {
30    context State connected:
31      "Set{self}−>closure(s | Transition.allInstances()
32        −>select(t | t.tar=s)−>collect(src))
33        −>exists(s | s.isInitial)"
34  }
```

Listing 10: WODEL program used to generate automaton exercises

In Listing 10, the first block (lines 6–11) produces 1 mutant for each seed model. WODEL-EDU uses these mutants to generate a web page with one *alternative response* exercise per seed model. Each exercise shows randomly either the seed model (in which case is correct) or the mutant (in which case is incorrect), and students must answer whether the shown automaton is correct or not. Figure 13 shows a screenshot with one of the generated exercises of this kind.

The second block in Listing 10 (lines 14–16) generates 2 mutants from each seed model. Then, WODEL-EDU generates a web page with *multiple diagram choice* exercises, where each seed model is shown among its mutants, and students must identify which is the correct one. As an example, Figure 14 shows one of these exercises.

The third and fourth blocks (lines 19–26) generate the necessary mutants for exercises of type *multiple emendation choice*. These exercises show an incorrect automaton together with a list of emendations, and students are required to select the subset of emendations that would correct the automaton. The first block (lines 19–22) generates the incorrect automaton, and its registry of applied mutations is used to automatically generate the correct emendation options. The other block (lines 23–26) mutates the automata generated in the first block. However, these new mutants are discarded, and only the registry of applied mutations is used to generate the incorrect emendation options. In this example, the mutation commands in the two blocks are the same: they change the target state of a random transition, and then invert the value of attribute isFinal of a random state. This has the effect of generating similar correct and incorrect emendations, which increases the exercise difficulty. Figure 15 shows one of the generated exercises, where the

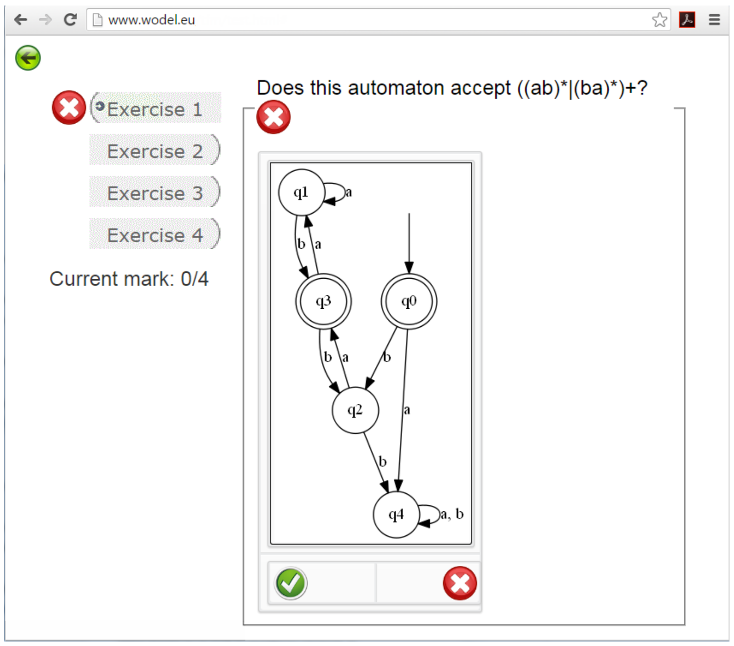


Figure 13: Screenshot of *alternative response* exercise

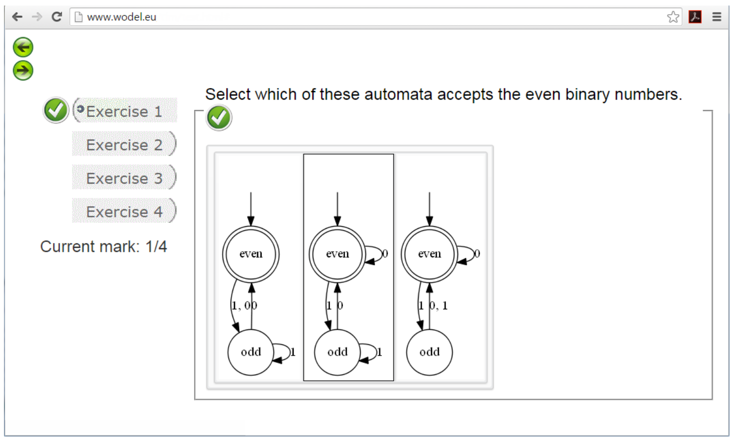
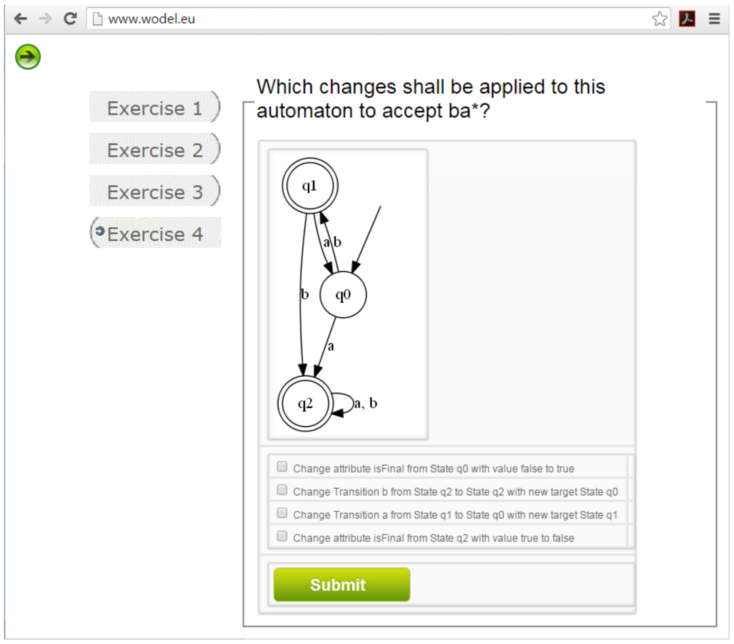


Figure 14: Screenshot of *multiple diagram choice* exercise

last two options should be selected to be correctly solved.

More in detail, the previous exercise is generated from the seed automaton shown in Figure 16a, which accepts the language "`ba*`". This automaton gets mutated as follows: state `q2` becomes final, and the transition from `q1` to `q1`



Which changes shall be applied to this automaton to accept ba*?

- Change attribute isFinal from State q0 with value false to true
- Change Transition b from State q2 to State q2 with new target State q0
- Change Transition a from State q1 to State q0 with new target State q1
- Change attribute isFinal from State q2 with value true to false

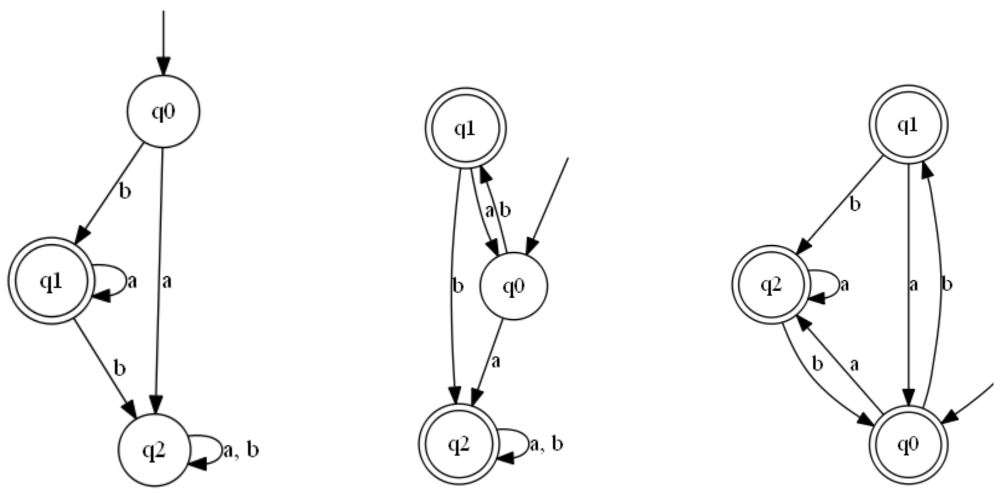
Figure 15: Screenshot of *multiple emendation choice* exercise

is changed so that it points to `q0`. Figure 16b shows the resulting mutant, which is the one shown in the exercise. Wodel-Edu uses the registry of applied mutations to generate the text of the correct emendations (the last two in Figure 15). These emendations, if applied on the mutant, would recover the original seed model. Then, the mutated automaton is mutated again, yielding the automaton in Figure 16c, where state `q0` has been made final and the transition `b` from `q2` to `q2` now points to `q0`. These mutations are used to generate two incorrect emendations in the exercise (the first two options in the list). Finally, Wodel-Edu presents the set of correct and incorrect emendations in random order.

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 explained in Section 5.1.

5.3. Evaluation of the quality of the generated exercises

In this section, we present the results of a preliminary user study whose aim is measuring the quality of the exercises automatically generated with



(a) Seed automaton (b) First mutated automaton used to produce correct emendations, and shown in the exercise (c) Second mutated automaton used to produce incorrect emendations

Figure 16: Steps to generate the exercise in Figure 15

Wodel-Edu. For this purpose, we have used Wodel-Edu to create a web application with three pages of exercises on finite automata: the first one contains exercises with multiple emendation choices, the second one contains exercises with multiple diagram choices, and the third one includes exercises with alternative response correct/incorrect. The exercises used in the evaluation are available at `http://www.wodel.eu/comlan16`.

There were 10 participants in the study, 8 men and 2 women, with age between 22 and 41 (31 years old in average). Five were computer science professors at the university, three were PhD students, one was a computer science student, and the last one worked as a computer science researcher in industry. Only one of them lacked training in automata theory.

The participants were asked to solve the exercises in the generated application with no time restrictions. Then, they were asked to rank each page of exercises (i.e., each kind of exercises) between 1 (completely disagree) and 5 (completely agree) with regard to the following questions:

- The exercise is easy to understand.
- The exercise has an appropriate level of difficulty.
- The exercise is useful to learn finite automata.

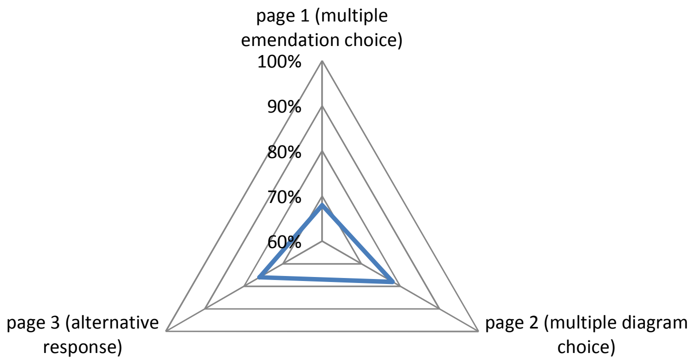
Optionally, they could also indicate the final grade obtained in each page of exercises, as well as provide comments and suggestions.

Among the comments, there were several suggestions related to the usability of the application user interface. In particular, several participants deemed the explanatory text in the exercises difficult to understand, and proposed using quotation marks around the regular expressions. Another participant complained that the exercises in the first page did not make clear that only the emendations making the automaton accept the words in the indicated language *and no other word* should be selected. In sight of these comments, we have modified our code generator to include more precise explanations in the exercises, and put quotation marks around the regular expressions. Anyhow, none of the received suggestions were related to the exercises themselves, but on their presentation.

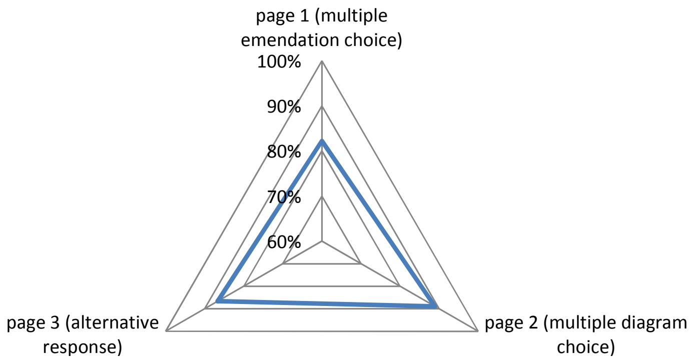
In addition, one participant discovered that the second exercise in the first page contained an emendation that could not be applied on the shown diagram. By inspecting the code, we discovered it was a bug in our code generator. This error, which was difficult to identify because it only occurred occasionally, has been fixed in the current version of Wodel-Edu.

Figure 17 shows the average of the scores given by the participants to the three pages of exercises in the three considered dimensions: understandability, difficulty and usefulness to learn automata. In Figure 17a, we can observe that the understandability of the exercises is not bad, but there is room for improvement. We have the lowest score (68%) in the first page which contains exercises with multiple emendation choices. The second and third pages of exercises have average scores 78% and 76% respectively. Unsurprisingly, the lowest score (1 out of 5) was given by the participant without training in automata theory. Regarding the difficulty of the exercises, Figure 17b shows that the first page of exercises is also considered the most difficult, with an average score of 82%. The other two pages have scores 89% (multiple diagram choice) and 86% (alternative response). Concerning the last dimension, Figure 17c shows that the participants considered the exercises very useful to learn automata, with scores 88% for the page with alternative response exercises, and 94% for the pages with multiple choices.

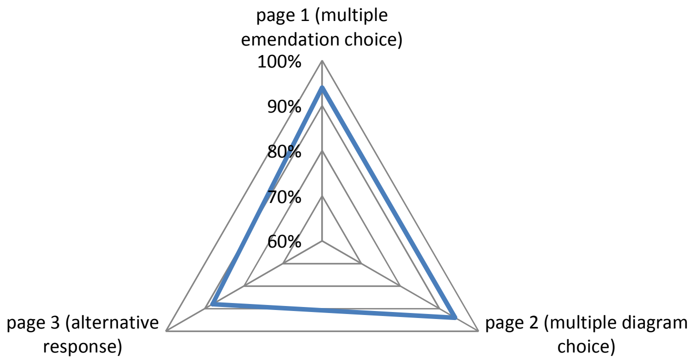
Figure 18 shows the average of the final grade obtained by the participants in the three pages of exercises, though only 60% of the participants provided their grade in the questionnaires. In average, the obtained final grade is above 60% for the second and third pages, while for the first page, the average grade is 50%. Two participants obtained the maximum grade in the first



(a) Average score to question *the exercise is easy to understand*



(b) Average score to question *the exercise has an appropriate level of difficulty*



(c) Average score to question *the exercise is useful to learn automata*

Figure 17: Average score in the pages of exercises considered in the evaluation

page, one participant obtained the maximum grade in the second page, and two obtained the maximum in the third page. No participant obtained the maximum grade in all three pages considered in the evaluation.

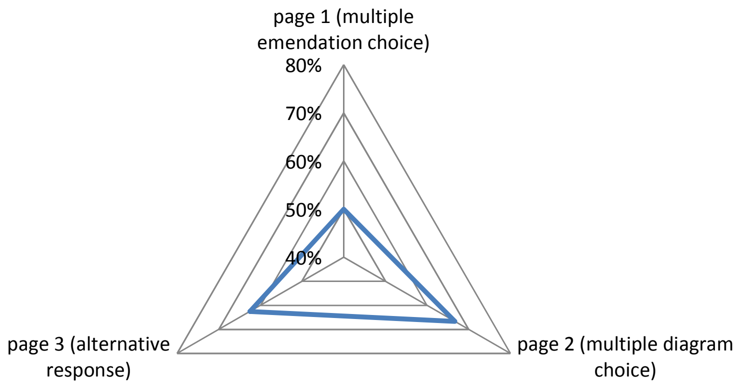


Figure 18: Average grade of the participants in the pages of exercises

We can extract several interesting conclusions from this evaluation. According to the results, the exercises with multiple emendation choices are difficult to understand. We believe one of the problems is that it might be difficult to represent mentally the result of applying several emendations to an automaton. One solution to this problem could be showing the automaton that would result after applying the set of selected emendations. Another option could be making the exercise interactive, allowing users to perform changes on the displayed automaton to build the correct solution. On the other hand, the participants considered that the exercises have an adequate level of difficulty, and they are very useful to learn automata. A last important issue is the need to provide a more precise description of the exercises, or including a tutorial. As proof of this, the participant that lacked training in automata theory gave the lowest score to the understandability of the three test exercises. All these ideas are very useful to help improving our tool and for future evaluations.

We also have to mention several threats to the validity of our evaluation. First, the order in which the exercises were presented to the participants may have influenced the perceived difficulty of the exercises. For instance, the first page of exercises which contained multiple emendation options is the most difficult, and this may have affected the perceived complexity of the second page of exercises. Another threat to the external validity is the fact that all our exercises are on finite automata; hence, the conclusions of the present user study may differ in case of considering exercises in other domains. Additionally, the evaluation only considers exercises automatically generated. To complement these results, in the future, we may conduct another user study where participants should solve exercises automatically generated and others made by hand. In this way, we could compare the results obtained in both cases, as well as evaluate whether the participants are able to discern which exercises were generated and which ones were human-made. Concerning the generality of the results across people, the participants in this study were not current or recent students of courses on automata theory, and one of them had not received training on this topic before. Thus, the results may be different in case the participants were receiving training on automata. The experiment was performed with only 10 participants, and so, another experiment with more participants is being planned. Finally, an evaluation of the tool from the professor perspective who is in charge of designing the exercises remains as future work.

6. Related Research

In this section, we revise related works on the two main lines of related research: the application of mutation techniques to different contexts and domains, and the usage of generative techniques in the educational domain.

6.1. Mutation techniques, languages and applications

Mutation is used in areas like model-based testing [22, 23, 24], model transformation testing [10], program testing [25, 26, 27, 28], adaptive systems [13], embedded systems [29], evaluation of clone-detection algorithms [30], generation of large model sets [31], education [11], or evolutionary algorithms [32, 33]. While most of these systems are built ad-hoc for a specific domain, Wodel may help in automating their construction. Next, we review approaches related to Wodel in different domains.

The mutation framework in [10] is specific to model transformation testing. It provides a set of predefined mutation operators which are transformation-specific and are defined with the Kermeta general-purpose model management language[4]. Instead, Wodel has mutation-specific primitives, and is not restricted to the mutation testing domain. Also for mutation testing,

[4]`http://www.kermeta.org/documents/`

the language MuDeL [34] allows describing mutation operators for grammar-based artefacts, typically programs. While MuDeL is based on replacement patterns, Wodel is based on operations. Moreover, Wodel has further facilities to combine mutation operators, apply them several times, and discard malformed or duplicated mutants. In the area of mutation-based testing, the system Muta-Pro [28] uses techniques to detect equivalent mutants, whereas we use model comparison. The mutation operators in [25] are specific to testing Android apps.

Model-based mutation testing has been applied to adaptive systems [13], model-based delegation security policies [27], logic formula [12] and software product lines [24]. It has also been used with Simulink models to compare clone-detection algorithms [30]. 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 [31] allows creating large models with some statistical properties. These models can then be used as tests for model management tools, e.g., for model transformation and matching. While the authors only illustrate creation operations, the creation context can be selected via stochastic properties. Similar to our registry of applied mutations, SiDiff [35] produces a history of model changes, though of lower-level nature than our mutation primitives. 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 also provides primitives for deletion and modification.

Mutation is also central in evolutionary algorithms, where problems are solved by generating a set of candidate solutions, which is iteratively improved by applying crossover and mutation operators. Candidate solutions are usually encoded as bit arrays, though some recent works [33] propose model-based approaches where the domain is expressed as a meta-model, the candidate solutions as models, and the mutation operators as model mutators. In particular, in [33], the authors define three mutation operators (add instance, remove instance and change weight) which are implemented using a general-purpose programming language. Instead, Wodel could have been used to create these and other operators.

In summary, our proposed DSL Wodel is novel as current mutation-

[5]`https://code.google.com/a/eclipselabs.org/p/ecore-mutator/`

based systems are commonly built by hand. The few existing languages to define mutations [10] focus on testing and work over grammars. As many applications need to specify and produce mutants, an extensible approach like ours is useful.

6.2. Automatic generation of exercises

Next, we revise works on the automatic generation and assessment of exercises, being most of them in the programming domain.

Petcha [36] is a tool that provides automatic assistance on teaching programming exercises. Its goal is to increase the number of programming exercises effectively solved by students. The tool can be used by professors to create exercises, and by students to solve them. Creating an exercise requires providing several resources such as the exercise description, test cases, correctors and feedback files. Then, instead of providing its own environment for solving exercises, Petcha relies on existing IDEs and e-learning systems to automate the exercise evaluation.

JExercise [37] is an Eclipse plugin to facilitate students testing their exercises. It is based on specifying the expected behaviour of classes and methods, a set of JUnit tests to verify the code, and a model of the solution.

In [38], the authors present a web-based tool that is able to automatically correct and provide feedback of UML diagram exercises. To define an exercise, professors must provide a description of the problem to be solved and the set of its admissible solutions. When a student provides a solution to the exercise, this gets compared with the set of correct solutions stored in the exercise definition. If no correct solution coincides with the one of the student, a feedback module selects the solution which is most similar to the one proposed by the student, and elaborates an appropriate feedback.

More similar to our application Wodel-Edu, in [11], the authors use mutation to generate exercises for state machines for a massive open online course. However, [11] 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.

Altogether, the analysed environments for the automatic generation of exercises are domain-specific (e.g., programming, UML diagrams or state machines). Hence, a framework like Wodel-Edu which allows the domain-independent automatic generation of exercises is novel.

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 as it can be applied to models of any meta-model, and its development environment can be extended for different applications. In this work, we showed an application consisting in a domain-independent framework for the generation of exercises based on the mutation of a correct solution. Although we have illustrated the framework by generating exercises on finite automata, it can also be used to generate exercises in other domains (e.g., UML class diagram or electronic circuits).

In our preliminary user study of Wodel-Edu, the participants considered that the generated exercises are useful in automata training. However, some aspects related to the usability of the web application user interface and the understandability of the exercises should be improved, in particular for the exercises with multiple emendation options. The evaluation has also been useful to identify some issues that should be taken into account in future evaluations or in real uses of the generated exercises, namely: the exercises should be listed in increasing order of complexity (first the easiest ones); in exercises with multiple emendation options, it is easier for students if all correct emendation options are grouped in a single checkbox; and exercises should include a clear description of what is required from the student.

We are currently working in enriching the languages of the Wodel-Edu plugin to support more complex learning environments (e.g., including gamification), exercises (e.g., interactive exercises where students have to correct an incorrect solution), and platforms (e.g., by enabling the resolution of exercises from mobiles devices and tablets [39], or for their integration with learning environments like Moodle[6]). Currently, modelDraw only permits the representation of graph-like visual languages, but in the future, we may extend it for other kinds of visual languages like Nassi-Shneiderman diagrams. We are also extending Wodel with support for mutation libraries, and the option to generate mutants which do not conform to the seed meta-model (e.g., negating the meta-model OCL invariants or violating reference cardinalities). We would also like to explore graphical ways to specify or represent parts of Wodel programs, like mutation block dependencies.

In the long term, we plan to develop Wodel post-processors for other

[6]`https://moodle.org/`

areas like model-based testing and evolutionary computation, which might trigger improvements in Wodel itself. We also plan to perform new user studies with more participants, where we will compare the results obtained in exercises generated with Wodel-Edu and exercises made by hand.

Acknowledgements. Work supported by the Spanish Ministry of Economy and Competitivity (TIN2014-52129-R) and the Madrid Region (S2013/ICE-3006).

References

[1] M. Brambilla, J. Cabot, M. Wimmer, Model-Driven Software Engineering in Practice, Morgan & Claypool, USA, 2012.

[2] A. R. da Silva, Model-driven engineering: A survey supported by the unified conceptual model, Computer Languages, Systems & Structures 43 (2015) 139–155.

[3] E. A. Marand, E. A. Marand, M. Challenger, DSML4CP: A domain-specific modeling language for concurrent programming, Computer Languages, Systems & Structures 44 (2015) 319–341. doi:10.1016/j.cl.2015.09.002.

[4] H. Prähofer, R. Schatz, C. Wirth, D. Hurnaus, H. Mössenböck, Monaco - A domain-specific language solution for reactive process control programming with hierarchical components, Computer Languages, Systems & Structures 39 (3) (2013) 67–94. doi:10.1016/j.cl.2013.02.001.

[5] A. Popovic, I. Lukovic, V. Dimitrieski, V. Djukic, A DSL for modeling application-specific functionalities of business applications, Computer Languages, Systems & Structures 43 (2015) 69–95. doi:10.1016/j.cl.2015.03.003.

[6] D. M. Groenewegen, E. Visser, Integration of data validation and user interface concerns in a DSL for web applications, Software and System Modeling 12 (1) (2013) 35–52. doi:10.1007/s10270-010-0173-9.

[7] H. B. Saritas, G. Kardas, A model driven architecture for the development of smart card software, Computer Languages, Systems & Structures 40 (2) (2014) 53–72. doi:10.1016/j.cl.2014.02.001.

[8] L. M. Rose, M. Herrmannsdoerfer, S. Mazanek, P. V. Gorp, S. Buchwald, T. Horn, E. Kalnina, A. Koch, K. Lano, B. Schätz, M. Wimmer, Graph and model transformation tools for model migration - empirical results from the transformation tool contest, Software and System Modeling 13 (1) (2014) 323–359. `doi:10.1007/s10270-012-0245-0`.

[9] T. Mens, P. V. Gorp, A taxonomy of model transformation, Electr. Notes Theor. Comput. Sci. 152 (2006) 125–142.

[10] V. Aranega, J. Mottu, A. Etien, T. Degueule, B. Baudry, J. Dekeyser, Towards an automation of the mutation analysis dedicated to model transformation, Software Testing, Verification & Reliability 25 (5-7) (2015) 653–683. `doi:10.1002/stvr.1532`.

[11] D. Sadigh, S. A. Seshia, M. Gupta, Automating exercise generation: A step towards meeting the MOOC challenge for embedded systems, in: Proc. Workshop on Embedded Systems Education (WESE), ACM, 2013, pp. 2:1–2:8.

[12] C. Henard, M. Papadakis, Y. L. Traon, Mutalog: A tool for mutating logic formulas, in: Proc. International Conference on Software Testing, Verification and Validation Workshops (ICSTW), IEEE CS, 2014, pp. 399–404.

[13] A. Bartel, B. Baudry, F. Munoz, J. Klein, T. Mouelhi, Y. L. Traon, Model driven mutation applied to adaptative systems testing, in: Proc. Mutation Analysis Workshop, 2011, pp. 408–413.

[14] P. Gómez-Abajo, E. Guerra, J. de Lara, Wodel: a domain-specific language for model mutation, in: Proc. Symposium on Applied Computing (SAC), ACM, 2016, pp. 1968–1973. `doi:10.1145/2851613.2851751`.

[15] D. Steinberg, F. Budinsky, M. Paternostro, E. Merks, EMF: Eclipse Modeling Framework, 2^{nd} Edition, Addison-Wesley Professional, 2008.

[16] H. Ehrig, K. Ehrig, U. Prange, G. Taentzer, Fundamentals of Algebraic Graph Transformation, Monographs in Theoretical Computer Science. An EATCS Series, Springer, 2006.

[17] P. Ammann, J. Offutt, Introduction to Software Testing, 1st Edition, Cambridge University Press, New York, NY, USA, 2008.

[18] M. Marcotty, H. Ledgard, G. V. Bochmann, A sampler of formal definitions, ACM Comput. Surv. 8 (2) (1976) 191–276. doi:10.1145/356669.356672.

[19] J. H. Andrews, Y. Zhang, General test result checking with log file analysis, IEEE Transactions on Software Engineering 29 (7) (2003) 634–648. doi:10.1109/TSE.2003.1214327.

[20] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, C. Zapf, An experimental determination of sufficient mutant operators, ACM Transactions on Software Engineering Methodology 5 (2) (1996) 99–118. doi:10.1145/227607.227610.

[21] E. Clayberg, D. Rubel, Eclipse Plug-ins, 3^rd Edition, Addison-Wesley Professional, 2008.

[22] B. K. Aichernig, H. Brandl, E. Jöbstl, W. Krenn, UML in action: a two-layered interpretation for testing, ACM SIGSOFT Software Engineering Notes 36 (1) (2011) 1–8. doi:10.1145/1921532.1921559.

[23] X. Devroey, G. Perrouin, P. Schobbens, P. Heymans, Poster: Vibes, transition system mutation made easy, in: Proc. International Conference on Software Engineering (ICSE), IEEE Computer Society, 2015, pp. 817–818. doi:10.1109/ICSE.2015.263.

[24] H. Lackner, M. Schmidt, Towards the assessment of software product line tests: a mutation system for variable systems, in: Proc. Workshop on Software Product Line Analysis Tools (SPLat), ACM, 2014, pp. 62–69. doi:10.1145/2647908.2655968.

[25] L. Deng, J. Offutt, P. Ammann, N. Mirzaei, Mutation operators for testing Android apps, Information and Software Technology In press. doi:10.1016/j.infsof.2016.04.012.

[26] J. P. Galeotti, C. A. Furia, E. May, G. Fraser, A. Zeller, Inferring loop invariants by mutation, dynamic analysis, and static checking, IEEE Transactions on Software Engineering 41 (10) (2015) 1019–1037.

[27] P. H. Nguyen, M. Papadakis, I. Rubab, Testing delegation policy enforcement via mutation analysis, in: Proc. International Conference

on Software Testing, Verification and Validation Workshops (ICSTW), 2013, pp. 34–42.

[28] A. M. R. Vincenzi, A. da Silva, M. E. Delamaro, J. C. Maldonado, Muta-Pro: Towards the definition of a mutation testing process, J. Braz. Comp. Soc. 12 (2) (2006) 49–61.

[29] N. Bombieri, F. Fummi, V. Guarnieri, G. Pravadelli, Testbench qualification of SystemC TLM protocols through mutation analysis, IEEE Transactions on Computers 63 (5) (2014) 1248–1261. `doi:10.1109/TC.2012.301`.

[30] M. Stephan, M. H. Alalfi, A. Stevenson, J. R. Cordy, Using mutation analysis for a model-clone detector comparison framework, in: Proc. International Conference on Software Engineering (ICSE), IEEE / ACM, 2013, pp. 1261–1264.

[31] P. Pietsch, H. S. Yazdi, U. Kelter, Controlled generation of models with defined properties, in: Proc. Software Engineering (SE), Vol. 198 of LNI, GI, 2012, pp. 95–106.

[32] J. Krall, T. Menzies, M. Davies, GALE: geometric active learning for search-based software engineering, IEEE Transactions on Software Engineering 41 (10) (2015) 1001–1018.

[33] A. Moawad, T. Hartmann, F. Fouquet, G. Nain, J. Klein, J. Bourcier, Polymer - A model-driven approach for simpler, safer, and evolutive multi-objective optimization development, in: Proc. International Conference on Model-Driven Engineering and Software Development (MODELSWARD), SciTePress, 2015, pp. 286–293.

[34] A. da Silva, J. C. Maldonado, MuDeL: a language and a system for describing and generating mutants, J. Braz. Comp. Soc. 8 (1) (2002) 73–86.

[35] S. Wenzel, Unique identification of elements in evolving software models, Software and System Modeling 13 (2) (2014) 679–711.

[36] R. A. P. Queirós, J. P. Leal, PETCHA: a programming exercises teaching assistant, in: Proc. Annual Conference on Innovation and Technology

in Computer Science Education (ITiCSE), ACM, 2012, pp. 192–197. doi:10.1145/2325296.2325344.

[37] H. Trætteberg, T. Aalberg, JExercise: a specification-based and test-driven exercise support plugin for Eclipse, in: OOPSLA workshop on Eclipse Technology eXchange (ETX), ACM, 2006, pp. 70–74. doi:10.1145/1188835.1188850.

[38] J. Soler, I. Boada, F. Prados, J. Poch, R. Fabregat, A formative assessment tool for conceptual database design using UML class diagram, iJET 5 (3) (2010) 27–33.

[39] D. Vaquero-Melchor, A. Garmendia, E. Guerra, J. de Lara, Towards enabling mobile domain-specific modelling, in: Proc. International Conference on Software Technologies (ICSOFT), SciTePress, 2016, pp. 117–122.