

## RESEARCH ARTICLE

WILEY

# Automated generation and correction of diagram-based exercises for Moodle

Pablo Gómez-Abajo  | Esther Guerra  | Juan de Lara 

Department of Computer Science,  
Universidad Autónoma de Madrid,  
Madrid, Spain

## Correspondence

Pablo Gómez-Abajo, Department of  
Computer Science, Universidad  
Autónoma de Madrid, Madrid, Spain.  
Email: [pablo.gomez@uam.es](mailto:pablo.gomez@uam.es)

## Funding information

Comunidad de Madrid,  
Grant/Award Number: S2018/TCS-4314;  
Ministerio de Ciencia e Innovación,  
Grant/Award Numbers: PID2021-  
122270OB-I00, TED2021-129381B-C21

## Abstract

One of the most time-consuming task for teachers is creating and correcting exercises to evaluate students. This is normally performed by hand, which incurs high time costs and is error-prone. A way to alleviate this problem is to provide an assistant tool that automates such tasks. In the case of exercises based on diagrams, they can be represented as models to enable their automated model-based generation for any target environment, like web or mobile applications, or learning platforms like MOODLE. In this paper, we propose an automated process for synthesizing five types of diagram-based exercises for the MOODLE platform. Being model-based, our solution is domain-agnostic (i.e., it can be applied to arbitrary domains like automata, electronics, or software design). We report on its use within a university course on automata theory, as well as evaluations of generality, effectiveness and efficiency, illustrating the benefits of our approach.

## KEYWORDS

automata, automated generation of exercises, model mutation, model-driven engineering, MOODLE

## 1 | INTRODUCTION

Designing and creating exercises to evaluate students is one of the most time-demanding activities professors must accomplish. Exercises are frequently created by hand, and need to be adapted to the format of the target learning platform. Moreover, many times, numerous variations of an exercise need to be created, to produce personalized tests that can be used for online evaluation. This problem has been exacerbated by the current need to move face-to-face teaching to an online setting. Overall, the creation of such a variety of exercises, their deployment into online platforms, and their manual

correction is not only time-consuming, but also error-prone.

A methodology to facilitate the generation of exercises that can be automatically graded would highly reduce the effort required from teachers. In the case of exercises based on diagrams—for example, in the domains of automata, electronic circuits, software design or chemistry—these can be represented as models to benefit from the automation techniques provided by model-driven engineering (MDE) [6].

MDE is a software engineering approach that uses models as the main artifacts in all development stages. It automates the repetitive tasks required in software

This is an open access article under the terms of the Creative Commons Attribution-NonCommercial-NoDerivs License, which permits use and distribution in any medium, provided the original work is properly cited, the use is non-commercial and no modifications or adaptations are made.

© 2023 The Authors. *Computer Applications in Engineering Education* published by Wiley Periodicals LLC.

processes, while enabling to focus on the creative ones. Models are often built using domain-specific languages (DSLs) tailored to a domain [21]. In addition, MDE provides techniques for model manipulation. *Model mutation* is one such type of manipulation, consisting in the creation of variants (called *mutants*) from a set of input *seed* models. This technique has proven useful to automatically generate and correct exercises [8, 27]. The rationale is that we can devise exercise templates based on the combination of seed models and their mutants, and automate the process to generate and correct them. For example, one exercise template may take one seed model as the correct solution, and one of its mutants as a wrong answer.

This work builds on the DSL *WODEL* for model mutation developed by Gómez-Abajo and collaborators [8, 11]. *WODEL* can apply mutations to models in any target domain defined by a meta-model, and can be extended with post-processing applications. *WODEL-EDU* [8] is one such post-processing applications for the automatic generation and correction of exercises. Professors can use *WODEL-EDU* as an assistant tool to automatically generate exercises in a simple way. *WODEL-EDU* is domain-independent (i.e., it can generate exercises for any domain), it can produce exercises of three different kinds, and deploy them as a web application. However, *WODEL-EDU* has limitations regarding the types of exercises supported, the target platforms covered, and some steps in the process are manual.

In this article, we extend *WODEL-EDU* to enable the generation of exercises for the *MOODLE* learning platform [23], as this is one of the most widely used course management systems nowadays. Targeting *MOODLE* permits making available both the generated exercises and the rest of the course material in the same place. In addition, we have expanded *WODEL-EDU* with two new kinds of multiple-choice exercises, and the possibility to automate the exercise generation process using a seed model synthesizer. The latter facilitates further the task of the teacher, who does not even need to provide base models upon which the exercises are generated. We present an evaluation of the quality of the exercises generated by *WODEL-EDU* in a university course on automata theory, as well as assessments of the generality, effectiveness, and efficiency of *WODEL-EDU*, showing good results.

The rest of the article is organized as follows. Section 2 describes related work. Section 3 overviews our methodology and shows a running example. Section 4 describes the generation of the different exercise kinds. Section 5 presents the architecture of our solution. Section 6 presents tool support. Finally,

Section 7 reports on the evaluations, and Section 8 concludes, identifying lines of future work.

## 2 | RELATED WORK

Our approach generates exercises that can be automatically graded by mutating (i.e., *making incorrect*) the correct solutions of the exercises, expressed as diagrams. A salient feature of our approach is its domain independence, as the domain can be configured using a meta-model, and the process is controlled via a suite of DSLs. Next, we compare with related approaches.

The literature reports on three main approaches to automatically generate and correct exercises: (i) applying a comparison between the provided answer and the solution of the problem, (ii) in domains like programming, relying on execution, (iii) introducing changes to the solutions and creating the exercises based on these changes, *WODEL-EDU* uses the latter approach. In the next subsections, we discuss related works within the three categories.

### 2.1 | Comparison-based approaches

The comparison-based approaches use an algorithm to compare the answers and the solution to a problem. This is useful when the problem consists in the creation or edition of the complete answer in a free way. Instead, *WODEL-EDU* synthesizes wrong answers, which are used to produce several types of exercises.

The work presented in [5] uses algorithms to grade modeling exercises by comparing the answers provided by the students with the solution. Snoeck et al. [29] propose a didactical Computer-Aided Software Engineering (CASE)-tool to build modeling exercises. This tool provides feedback to help the students with the solutions they are creating. However, this solution does not provide automated grading, being more of a learning assistant than a fully functional tool to generate exercises.

The proposal by Hoggarth et al. [14] uses a CASE tool to automatically assess diagram exercises, although without automated grading. Thomas et al. [34, 35] analyse the diagrams by building their minimal meaningful units to determine their overall similarity with the solution models. Soler et al. [30] present a web-based tool capable of automatically correcting exercises and providing feedback on UML class diagrams using a comparison algorithm.

Foss et al. [7] propose AutoER, a web system to create and automatically evaluate UML database design diagrams. The system supports instructor-created and automatically generated questions with instructor-configurable marking. Similar to us, the instructor provides a solution and a descriptive text. Different from us, the student creates the solution by interacting with the text, by requesting the system add diagram elements rather than drawing them manually. The approach is limited to this one kind of exercise, and is not applicable to other domains.

Gong [12], Liu et al. [18], and Wong et al. [37] propose systems for the automated grading of programming language assignments. They are based on symbolic execution and dynamic program analysis, semantic comparison, and extraction of a set of matching rules from the program outputs, respectively. The proposal by Wang et al. [36] generates automated feedback for introductory programming exercises, based on a search for the closest solution to the answer, the detection of syntactical differences, and a calculation of the minimal reparation. This solution does not provide automated grading.

In the context of evaluating pseudocode for dynamic programming problems, Xia and Zilles [38] used context-free grammar to parse students responses, and then compare them with respect to the correct solutions given by the instructor. Responses are not given in free form, but guided by a syntax-directed editor. Instead of text, our approach is based on diagrams, we support 5 different exercise kinds, and it is domain independent.

Kotsiopoulos et al. [16] introduce DaST, an online platform for the automated generation and solving of Data Science exercises. Their proposal includes a default set of common data science algorithms that can be extended. The students can provide their own data sets or generate random ones and propose a solution for the given problem. Then, they can use DaST to apply the selected algorithm over the data set and check their solution with the provided graphical output.

Alur et al. [3] propose an algorithm to automatically grade deterministic finite automata constructions. Their proposal includes a description language called MOSEL that allows defining regular languages, their transformation into descriptions of finite automata, and vice versa.

## 2.2 | Execution-based approaches

In some particular domains—like programming—when answers involve code, this can be executed, and many approaches benefit from this fact.

There are extensive works aiming at producing feedback and guidance for students working on programming tasks [20], as well as on automated grading of programming exercises [2]. For example, CodeRunner [19] is a Moodle plugin that supports code exercises, the answers of which are sent to a server for automated grading via pre-defined test suites. Other similar approaches have been created, for example, to support the automated evaluation of microcontroller assignments [32]. Versatile test oracles [33] and markup languages have been created to facilitate using such automated grading tools [22]. Our approach to automatic grading is not based on execution (and comparison of test results) but on derivation of wrong solutions from correct ones. While the previous works only deal with automated grading, we also automate exercise generation.

In the context of block-based visual programming, Ahmed et al. [1] use mutation to generate visual programming tasks (mazes) similar to a previously solved exercise. The authors use symbolic execution over the mutated codes to generate the visual task. In a similar vein, Pădurean et al. [26] use a neuro-symbolic technique for the same purpose.

Also, within programming, another possibility is asking tracing questions about the execution of programs. Some approaches can generate these questions automatically [31] using a complexity metric to select code snippets of similar complexity. As we will see in Section 5.2, WODEL-EDU can automatically generate diagrams (i.e., solutions matching an exercise statement) of similar complexity by using size parameters and constraints.

## 2.3 | Modification-based approaches

The literature includes several works that use a similar approach to WODEL-EDU. Sadigh et al. [27] propose a methodology to produce different versions of a problem by applying mutation, and then automatically generate their solution by SAT solving. They generalize the problems into templates and apply this approach to model-based and real-time problems.

Papasalouros [25] describes a method to automatically generate exercises based on Horn clauses and applies it to the domain of Euclidean Geometry. This proposal defines a set of operators to move from the initial to the goal states. Thus, the task to solve the problem consists of finding the path toward the solution.

While these approaches target specific domains (real-time, geometry), WODEL-EDU is domain-independent. It provides a fully-functional solution ready to be used to generate exercises on any domain, and the possibility to

automate the entire process by a complete problem synthesis.

### 3 | METHODOLOGY AND RUNNING EXAMPLE

This section overviews our approach for the automated generation of exercises (Section 3.1) and introduces an illustrative running example (Section 3.2).

#### 3.1 | Overview of our methodology

Figure 1 shows the process of our model-based approach to automatically generate exercises for diagrammatic languages. We distinguish three roles in this process: *language engineer*, *professor* of the course, and *student*.

First, the language engineer specifies the ingredients of the diagrammatic language (e.g., automata, electric circuits, or class diagrams). This specification comprises the language abstract syntax, that is, a meta-model [6], and its graphical representation (Steps 1 and 2 in the figure). In addition, the engineer defines interesting mutation operators for the domain, that is, modifications to the elements of the language that will produce different model variations (Step 3). Our approach uses such variations to create several kinds of exercises, as explained in Section 4. Finally, the engineer defines the textual representation for the model elements and the mutation operators in the text options of the exercises. These steps of the process are supported by a family of DSLs that are described in Section 5.1.

Next, the professor of the course configures the exercises. This implies providing a set of sample

models of interest, which are mutated during the exercise generation process. These sample models can be either manually created by the professor (Step 5b) or automatically synthesized (Step 5a), as described in Section 5.2. The professor can also configure other features of the exercises, such as their kind (one among alternative response, multiple diagram choice, multiple emendation choice, match pairs choice, and missing word choice), their statements, and whether reattempts should be allowed (Step 6). This configuration is performed using another DSL within the family (EDUTest), explained in Section 5.1.5. This information is used to automatically generate an exercises questionnaire for MOODLE (Step 7).

Finally, the students perform the questionnaire of exercises on MOODLE, which is automatically graded (Steps 8 and 9 of the figure).

#### 3.2 | Running example

In MDE, the syntax of a language is defined by a meta-model. This contains the classes, properties and relations of the elements of the language. A model belongs to a language if it conforms to the language meta-model. As a running example, we are creating exercises for deterministic finite automata (DFA). Figure 2 shows the meta-model of its syntax.

A model conforming to this meta-model is composed of an *Automaton* object with a name and an alphabet of *Symbols*. The *Automaton* contains a set of *States*, which have a name and can be initial and final, and a set of *Transitions*, which have a symbol and refer to a source (*src*) and a target (*tar*) *States*.

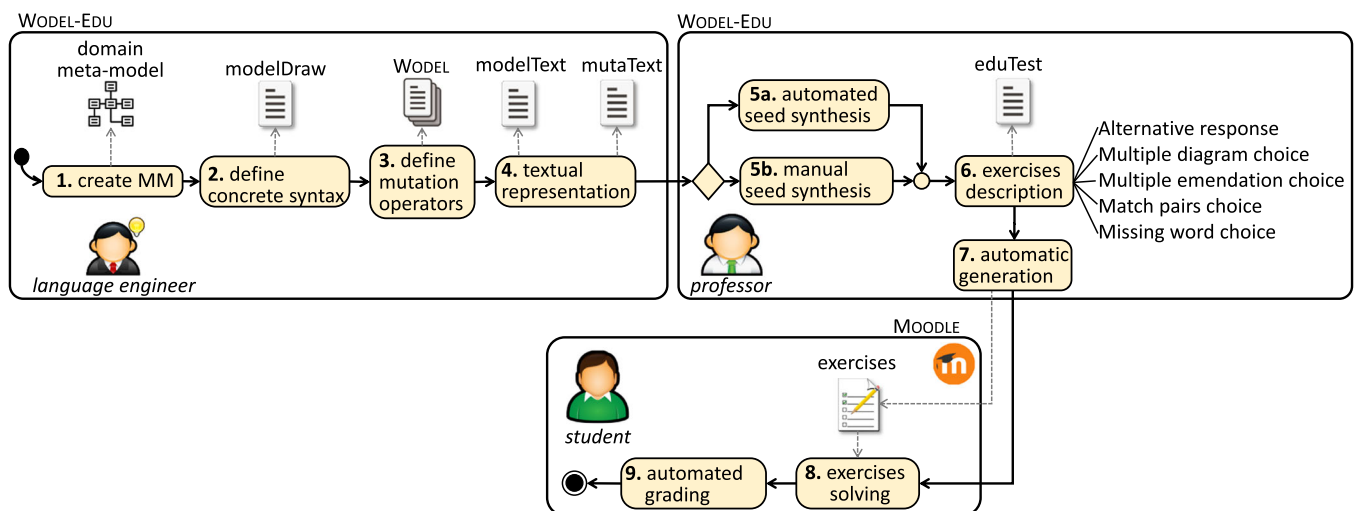


FIGURE 1 Our model-based process to automatically generate exercises for MOODLE.

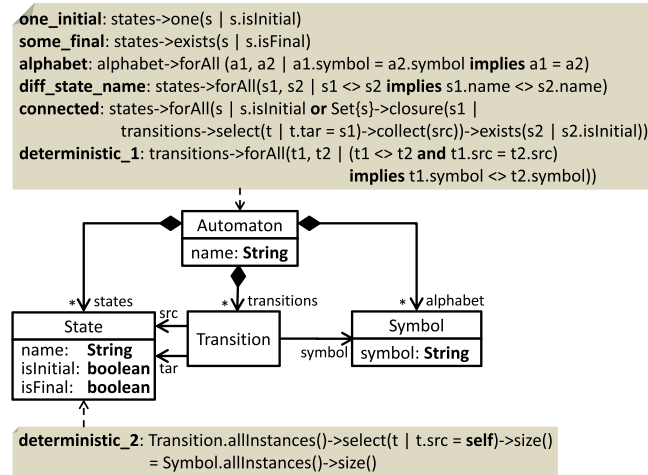


FIGURE 2 Meta-model of deterministic finite automata.

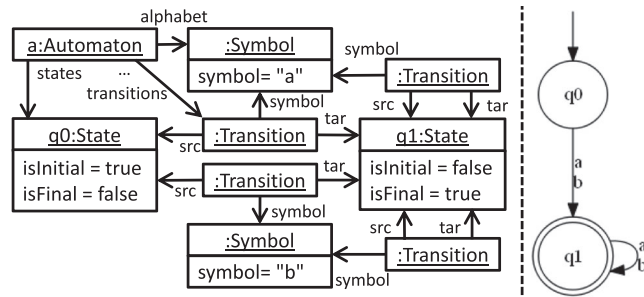


FIGURE 3 A valid deterministic finite automata (DFA) in abstract (left) and concrete (right) syntax.

Additional language requirements can be defined using OCL invariants [24]. These are expressions that need to evaluate to *true* for the model to be a valid member of the language. The meta-model in Figure 2 has 7 OCL invariants: 6 in class Automaton, and one in class State. The invariants *one\_initial* and *some\_final* ensure that the automaton has only one initial state and at least one final state. The invariant *alphabet* demands that the alphabet symbols are different. The invariant *diff\_state\_name* assures that each State has a different name. The invariant *connected* requires that there exists a path from the initial State to every other State in the automaton. The remaining invariants *deterministic\_1* and *deterministic\_2* ensure that the automaton is deterministic, that is, every State has exactly one output Transition for each symbol in the alphabet.

Figure 3 shows a DFA example conforming to this meta-model. The figure uses an abstract syntax to the left and the standard graphical representation for automata to the right (called concrete syntax [6]).

Figure 4 extends the DFA in Figure 3 with the transition shaded in red. This second automaton does not conform to

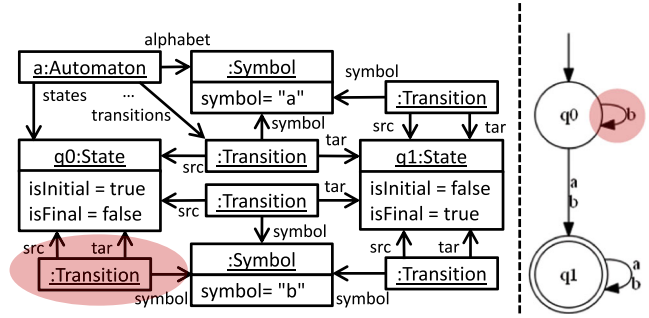


FIGURE 4 A deterministic finite automata (DFA) that does not conform to the meta-model in Figure 2.

the meta-model since it violates the invariants *deterministic\_1* (q0 has two outgoing transitions with symbol b) and *deterministic\_2* (q0 has more outgoing transitions than symbols in the alphabet).

## 4 | GENERATING EXERCISES BY MODEL MUTATION

This section introduces model mutation (Section 4.1) and presents the five kinds of exercises supported by our mutation-based proposal (Section 4.2).

### 4.1 | Model mutation

Model mutation is a kind of model transformation that produces a set of variants, called *mutant models*, from one or more input models, called *seed models*, by the application of one or more *mutation operators*. Model mutation has many applications, such as in transformation testing [4, 13] or in education [8, 27]. In the latter case, one can use models to represent a problem and its solution, and apply mutation to inject errors in the solution that the students must find.

Figure 5 shows the mutation of a model conforming to the meta-model of Figure 2. The seed model to the left represents a DFA that accepts even binary numbers. After applying to it a mutation operator that modifies the target of the shaded transition, we obtain the mutant model to the right, where the mutated transition is also shaded. The top half of the figure shows an excerpt of the abstract syntax of both models.

The DSL *WODEL* introduced in [8, 11] targets this kind of model mutations. Section 5 describes this DSL. We can exploit model mutation to generate diagram-based exercises. As an example, given the seed model in Figure 5, and the statement ‘Does this automaton accept the language of the even binary numbers?’, the generated exercise can alternatively present the diagram corresponding to this seed model



or to one of its mutants. The solution to the exercise (*true* or *false*) will depend on the model shown.

## 4.2 | Supported kinds of exercises

Our approach synthesizes five kinds of exercises based on model mutation: alternative response, multiple diagram choice, multiple emendation choice, match pairs choice, and missing word choice. Next, we explain and illustrate their generation schema.

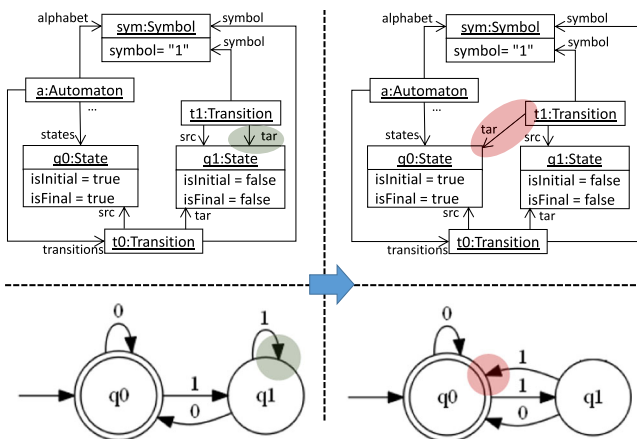


FIGURE 5 Example of model mutation applied to a deterministic finite automata (DFA).

### 4.2.1 | Alternative response

In this kind of exercise, the student must select whether the presented diagram corresponds to the one described by the exercise's statement. If the presented diagram corresponds to the seed model, it is correct, and if it corresponds to one of its mutants, it is wrong.

Figure 6a uses the running example to illustrate the strategy to generate this exercise kind. The process starts using the DFA at the top as the seed model, which accepts ' $a^*b^*$ '. Next, two mutation operators that modify the target of a transition are applied to this DFA. The middle of the figure shows two mutant examples generated by these mutation operators, where the affected elements are shaded. The bottom of the figure shows two generated exercises. The one at the bottom-left displays one of the mutants, along with the original statement. In this case, the correct answer is that the shown diagram does not match the statement. The bottom-right of the figure shows an exercise with the diagram of the seed model, and the solution is that the diagram matches the statement.

This and the following exercise kinds rely on the fact that the mutations change the language accepted by the DFA. As Section 5 explains, our approach supports specifying checks to test for model *equivalence*. This check discerns whether the seed model and the mutant are equivalent (i.e., they have different syntax but same semantics), and if so, the mutant is discarded. In the

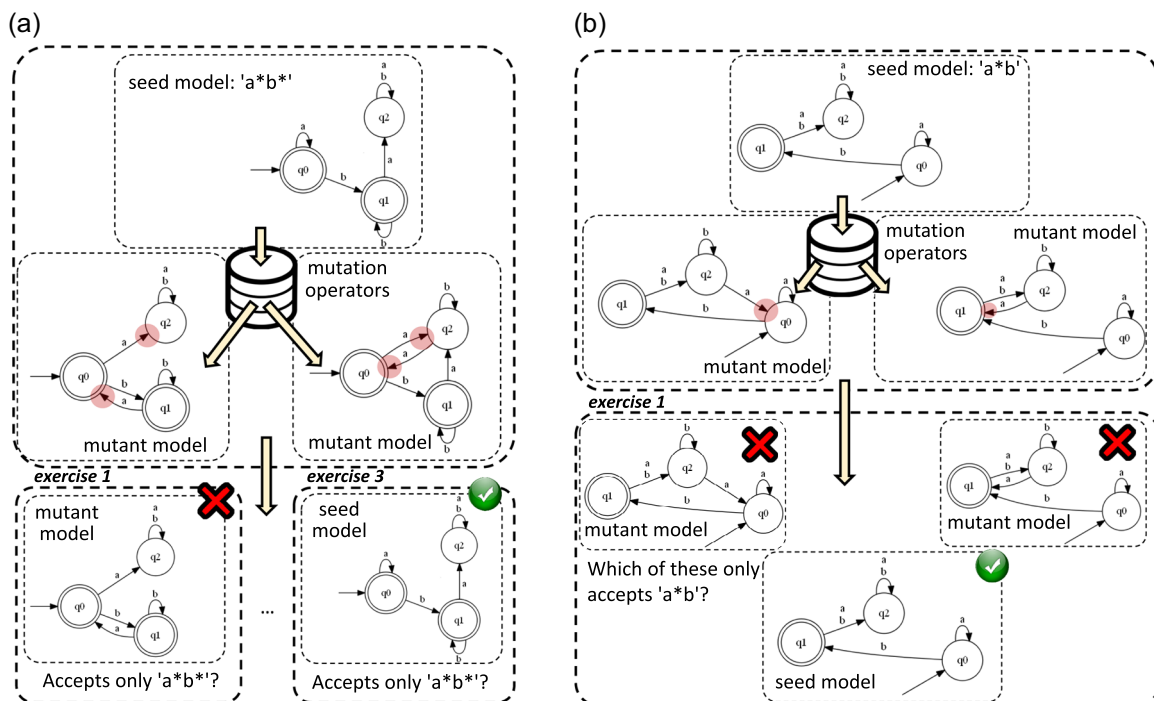


FIGURE 6 Schemas of the alternative response exercises (a) and multiple diagram choice exercises (b)

running example, the seed DFA and the mutant are considered equivalent if they accept the same language.

## 4.2.2 | Multiple diagram choice

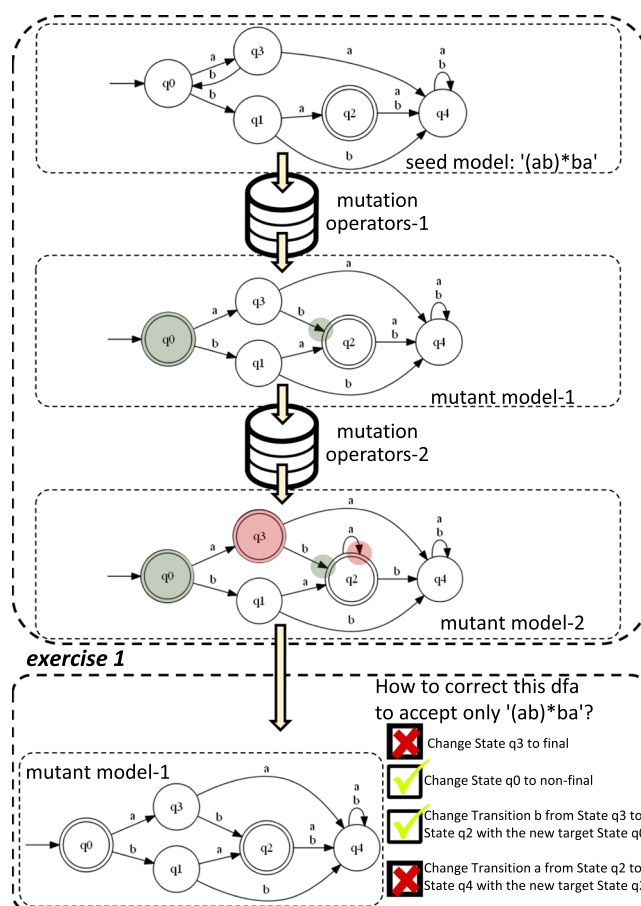
This kind of exercise presents a set of diagrams, and students must select the one matching the exercise's question. In this case, the solution corresponds to the diagram of the seed model, and the wrong answers correspond to its mutants.

Figure 6b illustrates the generation process. This starts from the seed model at the top, a DFA accepting ' $a^*b$ '. Next, a mutation operator that modifies the target of a transition is applied. The middle of the figure shows two of the resulting mutants, with the mutated elements shaded. The bottom of the figure displays the resulting exercise. It displays the diagrams corresponding to the generated mutants along with the diagram of the seed model. Solving the exercise consists in identifying the diagram that corresponds to the seed model.

## 4.2.3 | Multiple emendation choice

This exercise shows the diagram of a mutant model, which must be fixed with a set of text options to make it match the exercise's statement. While all options are applicable to the diagram, only a subset of them will properly emend it. The correct options are synthesized by reversing the mutation operators applied to generate the mutant shown in the exercise, and the wrong options are synthesized from other mutation operators applicable to this mutant.

Figure 7 illustrates the generation of this exercise kind. The process starts from the seed model at the top, a DFA accepting ' $(ab)^*ba$ '. In the second step, two mutation operators are applied to the seed model (mutation operators-1). The first operator changes a state to final, and the second modifies the target of a transition. The middle of the figure shows a mutant generated by these operators, with the affected elements shaded in green (mutant model-1). The third step applies a second set of mutation operators to the mutants obtained in the previous step (mutation operators-2). In the example, we have applied the same operators as in the first mutation step to make the exercise more challenging to solve, since this will generate text options that are similar to each other. The middle of the figure (mutant-model-2) shows a generated mutant after applying this second set of operators. It presents the elements affected by the previous mutation step shaded in green, and those affected by the current mutation step shaded in red.



**FIGURE 7** Schema of the multiple emendation choice exercises.

The bottom of the figure shows the resulting exercise. It presents the diagram corresponding to the mutant generated in the first mutation step, along with a statement asking which of the presented options emend the DFA to accept the initial language. These options are synthesized as follows: the text options that are solutions correspond to the mutation operators applied in the first step, and they are presented in a reverse order to undo the applied mutations; and the incorrect options are generated from the mutation operators applied in the second step, thus, they are applicable to the presented model, but when applied, they lead to a different model. As in previous cases, we make sure that the seed model and the mutants are not equivalent.

## 4.2.4 | Match pairs choice

This exercise kind presents a diagram and a set of statements that need to be matched with a list of text options. The statements are textual representations of the

mutants generated from the seed model, and each text option in the list is the textual representation of the applied mutation operators.

Figure 8a illustrates the synthesis of this kind of exercise. The process starts with a DFA accepting  $'((a|b)b^*a)^*'$ , and applying to it a mutation operator, which in this case modifies the target of a transition. The middle of the figure shows three of the generated mutants with the mutated elements shaded in red. Finally, the generated exercise at the bottom shows the diagram corresponding to the seed model, and for each generated mutant, it presents a statement that describes the mutant. These statements are obtained from the mutants by means of a model-to-text synthesizer described in Section 5. In the domain of automata, these statements are the regular expressions that define the language accepted by the DFA (in this example, the mutants accept  $'a^*|(a^*bb^*a)^*'$ ,  $'((a|b)|(a|b))^*'$ , and  $'b^*|(b^*ab^*a)^*'$ ). For each statement, the exercise presents a drop-down list with a description of the applied mutation operators in random order. The student must match the statements that identify each mutant with the description of the mutation operators used to generate them.

#### 4.2.5 | Missing word choice

This kind of exercise presents the diagram of a generated mutant and a set of text statements with gaps, which the student must fill. The statements are generated by reversing the applied mutation operators. The gaps in the statements can be filled with text options available in drop-down lists. These options correspond to the values that each variable may have in the mutation operators. The student must select one option for each drop-down list. The solutions correspond to the values that these variables took in the applied mutation operators.

Figure 8b illustrates the strategy followed. The process starts from the seed model at the top, to which a set of mutation operators is applied. In this example, the set includes an operator that modifies the target of a transition, and another that changes a state to final. The middle of the figure shows one generated mutant with the affected elements shaded in red. The generated exercise at the bottom shows the diagram of the mutant along with texts describing the reversal of the applied mutation operators. These texts contain drop-down lists for each variable in the mutation,

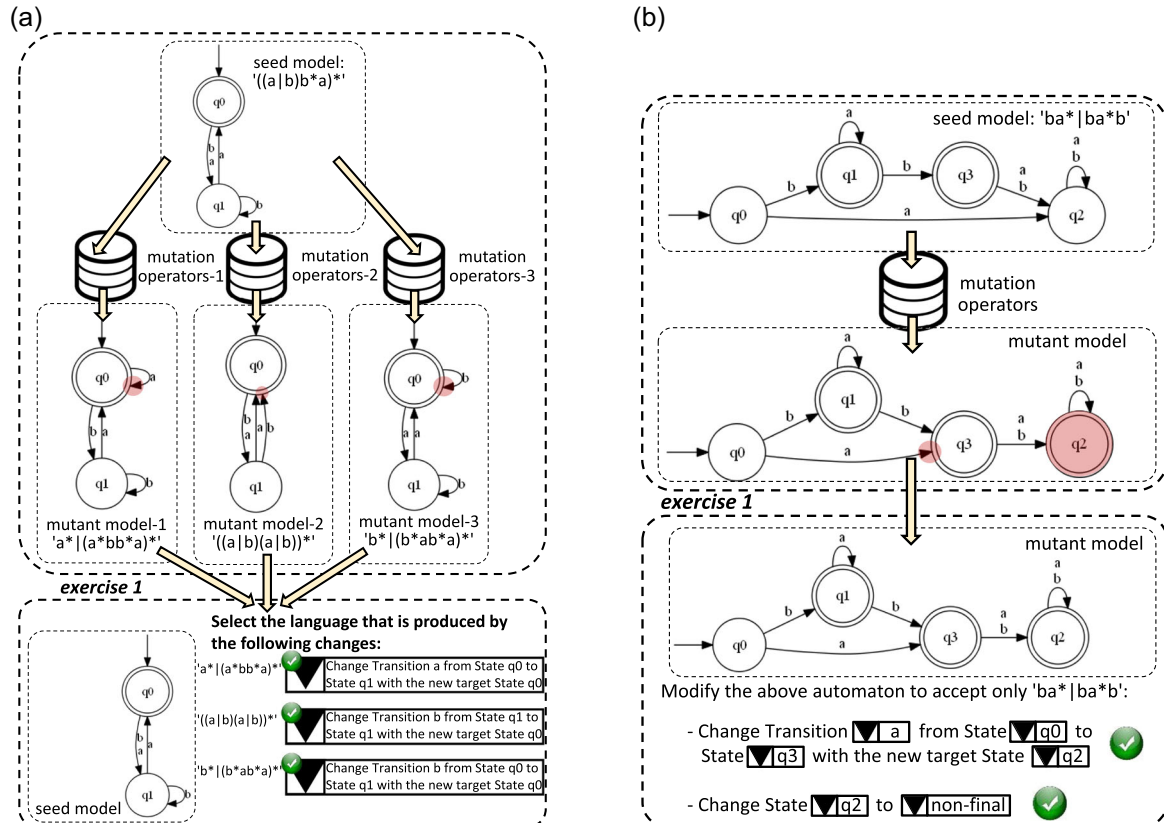


FIGURE 8 Schemas of match pairs choice (a) and missing word choice exercises (b).



listing the values that such variables may take, for example, ‘Change state ( $q_0$ ,  $q_1$ ,  $q_2$ , and  $q_3$ ) to (initial, non-initial, final, and non-final)’. The student must identify the values that these variables take in the applied mutation operators. This corresponds to the modifications needed in the DFA to accept the language in the statement.

## 5 | ARCHITECTURE

To automate the generation of diagram-based exercises using mutation, we have extended the WODEL-EDU framework proposed by Gómez-Abajo and collaborators [8]. Our extension supports more types of exercises, their generation for the MOODLE platform, the possibility to specify checks to detect model equivalence, the generation of alternative text representations of models, and the automated synthesis of seed models for their use in exercise generation.

WODEL-EDU is designed as a post-processor of the WODEL model mutation engine [11]. It provides a family of DSLs to configure the exercise generation process. Figure 9 shows its component-based architecture.

First, as introduced in Figure 1, the language engineer has to provide: the language meta-model; the mutation operators of interest for the target domain, described using the WODEL DSL; a specification of the graphical and textual representation of the meta-model elements, using the MODELDraw and MODELText DSLs; and the textual representation of the applied mutation operators, using the MUTAText DSL. WODEL provides extension points for specifying alternative model representations that may require complex algorithms. For our running example, the engineer can use this extension to produce the regular expression of a DFA. In addition, WODEL provides another extension point to specify equivalence criteria. For example, the engineer can

implement an extension to detect whether two automata accept the same language.

In the second step, the professor provides the seed models from which WODEL will generate mutants by applying the mutation operators defined by the language engineer. These seed models can also be automatically generated by means of a configurable seed model synthesizer. Next, the professor can use the EDUTest DSL to configure the kind and format of the exercises to be generated. From this configuration, WODEL-EDU generates the exercises for the target environment, either as a stand-alone web application, or for MOODLE.

In this section, we introduce the DSLs that allow configuring the exercises (Section 5.1), describe how to use the seed model synthesizer to simplify the exercise generation process (Section 5.2), and illustrate how we integrate this approach into the MOODLE platform (Section 5.3).

### 5.1 | The WODEL-EDU DSLs

WODEL-EDU offers a family of DSLs to configure the generation of the exercises, which we explain next.

#### 5.1.1 | The Wodel DSL

The language engineer uses the WODEL DSL to encode the mutation operators [8, 11]. WODEL provides mutation primitives to manipulate model objects, such as `select`, `create`, `clone`, `modify`, `delete`, and `retype` (i.e., change the type of an object to a sibling type); and to `create`, `modify` and `delete` relations.

Listing 1 shows a WODEL program that defines a mutation operator for DFA. Line 1 declares the strategy for mutant generation, which in this case is a stochastic generation of a maximum of four mutants. Alternatively, WODEL provides an exhaustive mode, which generates all possible mutants for the given mutation operators. Line 2 specifies the output folder to store the mutants and the input folder containing the seed models. Line 3 declares the URI or location of the language meta-model. The rest of the program defines a mutation operator called `mts2nf`. Lines 7–9 select some elements, storing them in variables. Line 7 selects an initial State, which is stored in variable `s0`; line 8 a non-final State different from `s0`, that is, saved in variable `s1`; and line 9 a Transition from `s0`. Then, lines 10–11 swap the target State of the Transition in `t0` with

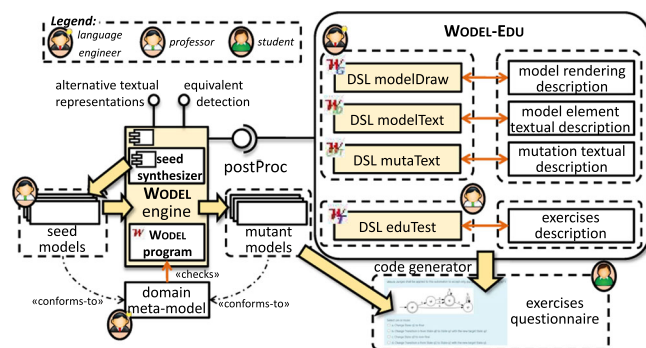


FIGURE 9 Architecture of WODEL-EDU.

the target State of another Transition that has the State in s1 as its target.

```

1 generate 4 mutants
2 in "out/" from "model/"
3 metamodel "http://dfa.com"
4
5 with blocks {
6   mts2nf {
7     s0 = select one State where {isInitial
8       = true}
9     s1 = select one State where {self <> s0 and
10       isFinal = false}
11     t0 = select one Transition where {src = s0}
12     modify one Transition where {self <> t0 and
13       tar = s1}
14   }
15 }

```

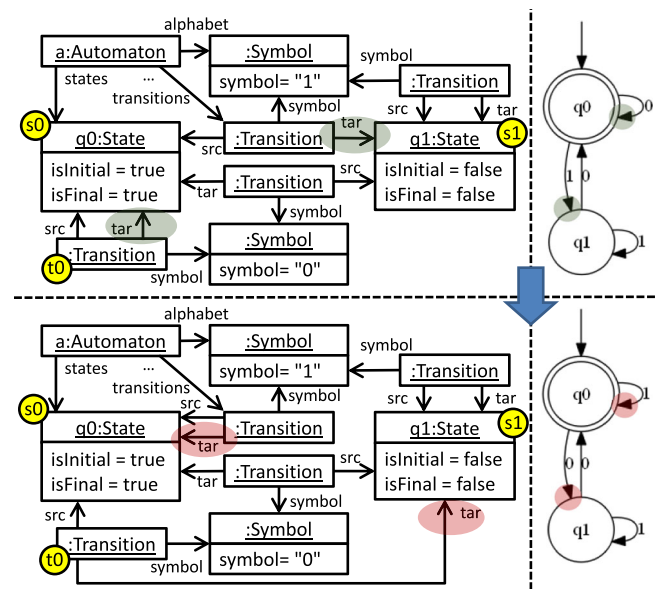
**LISTING 1** Defining a mutation operator with WODEL.

Figure 10 shows an example application of the defined mts2nf operator to an automaton. The model elements saved in memory during the operator execution are labeled with the name of the corresponding variables in Listing 1.

### 5.1.2 | The modelDraw DSL

The DSL MODELDraw allows configuring the graphical representation of the models, that is, the kind of node or connection representing each element in the language meta-model. Nodes support shapes like circles, ellipses, or rectangles. Connections support source and target decorations, like arrowheads or diamonds. Class attributes can be displayed as labels attached to the shapes and connections. For the running example, we display the state names inside a circle, and the transition symbols in the connections. MODELDraw supports conditional styles, where the representation of a class may change depending on the value of its attributes.

Listing 2 shows the MODELDraw specification for our example. The first line declares the language meta-model. Next, the listing declares that initial states are represented as marked circles (line 4), nonfinal states as circles (line 5), final states as double circles (line 6), and transitions as connections labeled with their symbols (line 7).



**FIGURE 10** Example application of the mts2nf operator.

```

1 metamodel "http://dfa.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
8     label=symbol.symbol
9 }

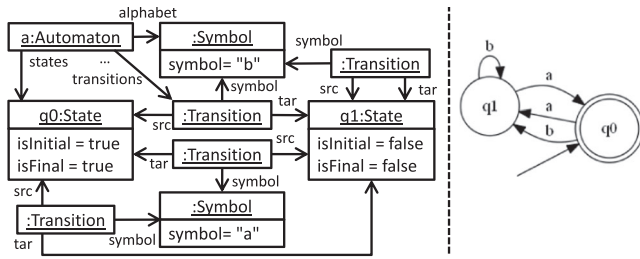
```

**LISTING 2** MODELDraw specification for the meta-model of Figure 2.

Figure 11 shows an example DFA, in its abstract syntax on the left, and graphically using this MODELDraw configuration on the right.

### 5.1.3 | The modelText DSL

The text options for the exercises *multiple emendation choice*, *match pairs*, and *missing words* need to represent the model elements textually. By default, when WODEL-EDU needs to generate a textual description of an element, this description corresponds to its attribute name if it exists, or to the class name otherwise. MODELText allows overriding this text representation for selected meta-model classes and relations. This is done by providing a text template where the expressions preceded by % are evaluated



**FIGURE 11** Deterministic finite automata (DFA) in abstract syntax (left) and graphical concrete syntax rendered using MODELDraw (right).

over the element, leading to the substitution of their value in the resulting string.

Listing 3 shows the MODELTEXT specification for the running example. Line 3 specifies that if an object of type State is used in some text, it will be written as “State” followed by its name. For example, “State q0” if the name of the state is “q0”. Lines 4–5 define prefixes for the state descriptions depending on the value of its attributes isFinal and isInitial. Hence, a state named q0 that is final, is textually represented as “final State q0”. If an element is both final and initial, both prefixes are conjoined. Line 6 defines that the objects of type Transition should be textually represented as “Transition” followed by the transition symbol. Finally, lines 7–8 declare that the references src and tar of Transition objects are to be referred to as “source” and “target”, respectively.

```

1 metamodel "http://dfa.com"
2
3 >State: State %name
4 >State(isFinal): final
5 >State(isInitial): initial
6 >Transition: Transition %symbol.symbol
7 >Transition.src: source
8 >Transition.tar: target

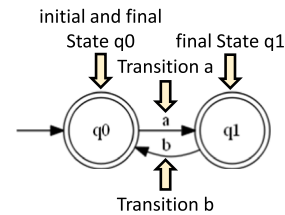
```

**LISTING 3** MODELTEXT specification for the running example.

Figure 12 shows a DFA with the name that each element takes according to the configuration in Listing 3.

### 5.1.4 | The mutaText DSL

Mutation operators also need to be represented as text. This is required to create the list of options in the



**FIGURE 12** Textual representation of a deterministic finite automata (DFA) using Listing 3.

multiple emendation choice, match pairs, and missing words exercises.

WODEL-EDU has default textual templates to represent each mutation primitive. These templates contain placeholders for the textual representation of the elements they are applied to. For example, the default template for the modify target reference primitive is ‘Change <refClassName> <refName> from <srcObjectName> to <tarObjectName> with the new target <newTarObjectName>’. MUTATEX allows overriding these default templates.

```

1 metamodel "http://dfa.com"
2
3 >TargetReferenceChanged:
4 Change %object from %fromObject to %
  toObject
5 with the new %refName %oldToObject /
6 Change %object from %fromObject to %
  oldToObject
7 with the new %refName %toObject
8
9 >AttributeChanged:
10 Change %object to %oldValue /
11 Change %object to %value

```

**LISTING 4** MUTATEX specification for the running example.

Listing 4 shows the MUTATEX specification for the running example. It defines the text to represent the modify reference and modify attribute operators both when it is displayed in a correct answer of an exercise (lines 4–5 and 10) and when it is a wrong answer (lines 6–7 and 11). In addition, it enables using predefined variables that contain information about the applied mutation, such as %object, which identifies the mutated object, or %refName, which has the name of the reference used in the mutation. These variables return the textual representation of the object or

reference specified with `MODELTEXT`, or a default textual representation if this specification is not available. For example, the text defined in line 10 of Listing 4 in combination with the `MODELTEXT` definition of Listing 3 generates text options such as: Change State `q1` to final.

### 5.1.5 | The eduTest DSL

This DSL allows configuring the following aspects of the exercises: kind (*alternative response*, *multiple diagram choice*, *multiple choice emendation*, *match pairs*, and *missing words*), order, statement, and retry options. From this specification, `WODEL-EDU` generates the exercises for the target environment (currently, either `MOODLE` or a stand-alone web application).

```

1  navigation=free
2
3  MissingWords mtsrfs1 {
4    retry=no
5    description for 'exercise1.model' =
6    'Complete the following text with the
      options that will modify the automata in
      order to only accept the language
      defined by ' %text('reg-exp')
7    description for 'exercise2.model' =
8    'Complete the following text with the
      options that will modify the automata in
      order to only accept the language
      defined by ' %text('reg-exp')
9  }
10
11 MatchPairs rfs1, mts2, rfs2 {
12   retry=no, text='reg-exp'
13   description for 'exercise3.model' =
14   'Match the correct option on the right that
       modifies the below automaton to
       correspond with each of the regular
       expressions on the left'
15   description for 'exercise4.model' =
16   'Match the correct option on the right that
       modifies the below automaton to
       correspond with each of the regular
       expressions on the left'
17 }
```

**LISTING 5** `EDUTEST` specification for the running example.

Listing 5 shows an `EDUTEST` specification with some exercises. Line 1 states that the exercises can be solved in

any order. Lines 3–8 define two exercises of kind missing words, which allow a single attempt (`retry=no`) and use the mutants generated by the `mtsrfs1` operator. Lines 5–8 specify the statement of the exercises generated from the seed models `exercise1.model` and `exercise2.model`, which may have been manually created by the professor, or as Section 5.2 shows, it can also be generated automatically. The statements text uses the inline variable `%text('reg-exp')`, which is automatically substituted by a regular expression defining the language the corresponding model accepts. Such alternative textual representations from models are generated by an implementation called *reg-exp* of the extension point provided by `WODEL` (see Figure 9).

Lines 11–16 define two exercises of kind *match pairs*, generated using the three mutation operators `rfs1`, `mts2`, `rfs2`. The professor of the course can input the variable `text='reg-exp'` (line 12) to generate the description of each exercise option using the extension point (cf. Figure 8a). This saves the effort to calculate the regular expression for each seed model, and is especially useful when using the automated synthesis of (a potentially large set of) seed models. Moreover, further implementations of the extension point can be used to define additional alternative representations, for example, to obtain the regular grammar equivalent to the automaton.

## 5.2 | Generation of exercises via seed model synthesis

The `WODEL` engine provides a model synthesizer to generate seed models from `WODEL` programs automatically (see Figure 9). Professors can benefit from this functionality to synthesize the seed models used to generate the exercises, hence achieving full automation of the exercise generation process.

Given a `WODEL` program, the synthesizer generates models over which all mutation operators in the program are applicable [9, 10]. The generation is based on model search, a technique that applies constraint resolution over models [28]. Several model finders are available for this task, like Alloy [15], the USE validator [17], or EFinder [28].

Figure 13 shows the process for automated seed model synthesis. The synthesizer enriches the language meta-model and its OCL invariants with additional OCL constraints derived from the `WODEL` program. These additional constraints encode the requirements a seed model must fulfill to allow the application of the mutation operators in the `WODEL` program. The enriched



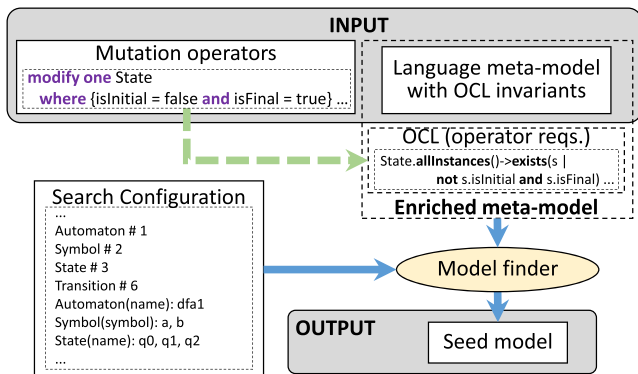


FIGURE 13 Automated model synthesis.

meta-model is loaded into the model finder, which performs a bounded search of the meta-model instances that satisfy all the OCL constraints. If such a model exists in the given search scope, then it is ensured that all the WODEL program statements are applicable over it.

As Figure 13 shows, our approach supports defining a search configuration customizing several features of the models to be generated, such as the number of objects of each class, the admissible values for string attributes, and additional OCL constraints that the models must fulfill. These configurations enable the synthesis of seed models with different complexity, which in our context is useful to generate exercises with different degrees of difficulty depending on the size of the generated DFA. For example, we may create a search configuration with low complexity that generates DFA with two alphabet symbols, two states, and four transitions; and another one with higher complexity by increasing these values.

The search configurations may include extra OCL constraints to control the shape of the generated models. For the running example, we may include constraints ensuring that the initial state is also final; to have several final states; or a sink (error) state. Moreover, the search can use an initial seed model. In such a case, the generated models include the initial model, which is useful to provide common patterns that each generated model should contain.

### 5.3 | Integration into the MOODLE platform

WODEL-EDU features an extension point that can be used to produce code to access the exercises on different platforms. For this work, we extended WODEL-EDU with a generator to synthesize code and deploy the defined

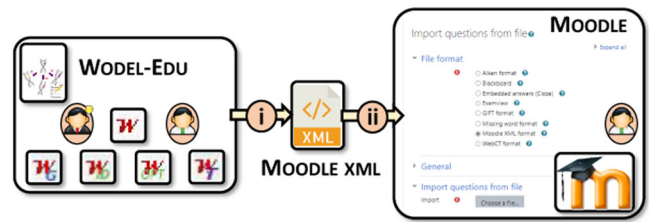


FIGURE 14 Generation of exercises for the MOODLE platform.

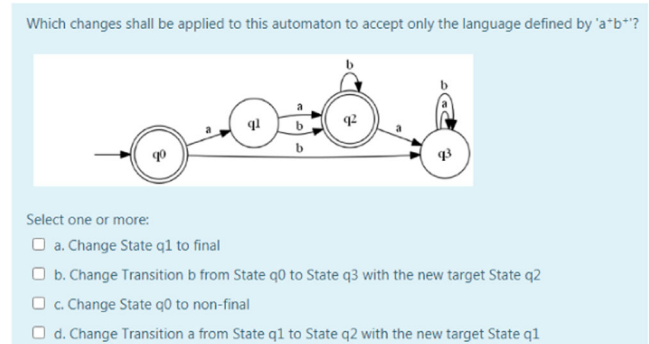


FIGURE 15 Generated emendation choice exercise for MOODLE.

exercises within MOODLE. However, the modular design of the extension point enables targeting other platforms like edX or Coursera. Such design has the advantage that the exact definition of exercises can be deployed on those different platforms.

As illustrated in Figure 14, the process used to integrate this approach into the MOODLE platform follows these two steps: (i) generation of the exercises in the MOODLE xml question bank format by using the generation mechanism provided by the WODEL-EDU tool, and (ii) import of these sets of exercises into the corresponding sections of the selected MOODLE course. As an example, Figure 15 shows an emendation choice exercise generated for MOODLE.

## 6 | TOOL SUPPORT

We have implemented the previous concepts in the WODEL-EDU tool, which is freely available as an Eclipse plugin in <https://gomezabajo.github.io/Wodel/wodel-edu.html>. This website contains video tutorials, and all models used in this article.

Figure 16a shows the WODEL-EDU environment. It provides editors for the five DSLs supporting the configuration of exercises and featuring rich



development functionality like syntax highlight and code completion. In the figure, Label 1 shows the project explorer containing the involved files. Label 2 shows the WODEL editor, with the mutation operators to be used for generating the exercises. Label 3 shows a table view with statistics on the number of times each meta-model element is affected by the application of the mutation operators. This view is complemented with the view labeled as 4, which depicts graphically the meta-model coverage by the mutation operators. Both views permit the language engineer to analyse the coverage of the meta-model by the mutator operators, and so to understand which parts of the models are never being mutated, or are mutated often. In the figure, we can observe that neither Symbol or Automaton objects are mutated, but the Transition and State objects are.

Figure 16b shows the first two pages of the wizard of the seed model synthesizer (cf. Section 5.2). The wizard enables the synthesis of seed models according to a given configuration. The first page of the wizard (Label 1) permits configuring the number of seed models to generate, and the mutation operators considered in the search. The second page (Label 2) declares the number of objects of each class to include in each generated model, and the labels used for the string attributes of these objects. One last page (not shown) enables specifying additional OCL constraints that the synthesized models must fulfill, and an initial model to start the search. In addition, the synthesizer wizard permits storing and loading these search configurations.

## 7 | EVALUATION

In this section, we present an evaluation aiming at measuring the quality of the exercises automatically generated with WODEL-EDU, as well as the generality, effectiveness, and efficiency of the tool. This way, our goal is to answer the following research questions (RQs):

**RQ1:** *Are the exercises generated with WODEL-EDU useful to learn the target domain?*

**RQ2:** *Are the generated exercises easy to understand?*

**RQ3:** *Is WODEL-EDU suitable to generate exercises for different domains?*

**RQ4:** *How effective and efficient is WODEL-EDU to generate exercises?*

For RQ1, we first consider the perceived usefulness to learn the target domain. Since mutations change the models automatically, and we also use automatically generated seed models, in RQ2, we want to assess whether the exercises are easily understood. For RQ3, we evaluate WODEL-EDU's generality by reporting on the generation of a set of exercises for a different domain: logic circuits (LCs). Finally, in RQ4, we evaluate the effectiveness (i.e., how many exercises can be automatically generated out of a certain number of seed models) and the efficiency (i.e., time performance) of WODEL-EDU for generating exercises for different types of automata and LCs.

In the following, Sections 7.1–7.3 report on the experiments designed for each RQ, their results, and

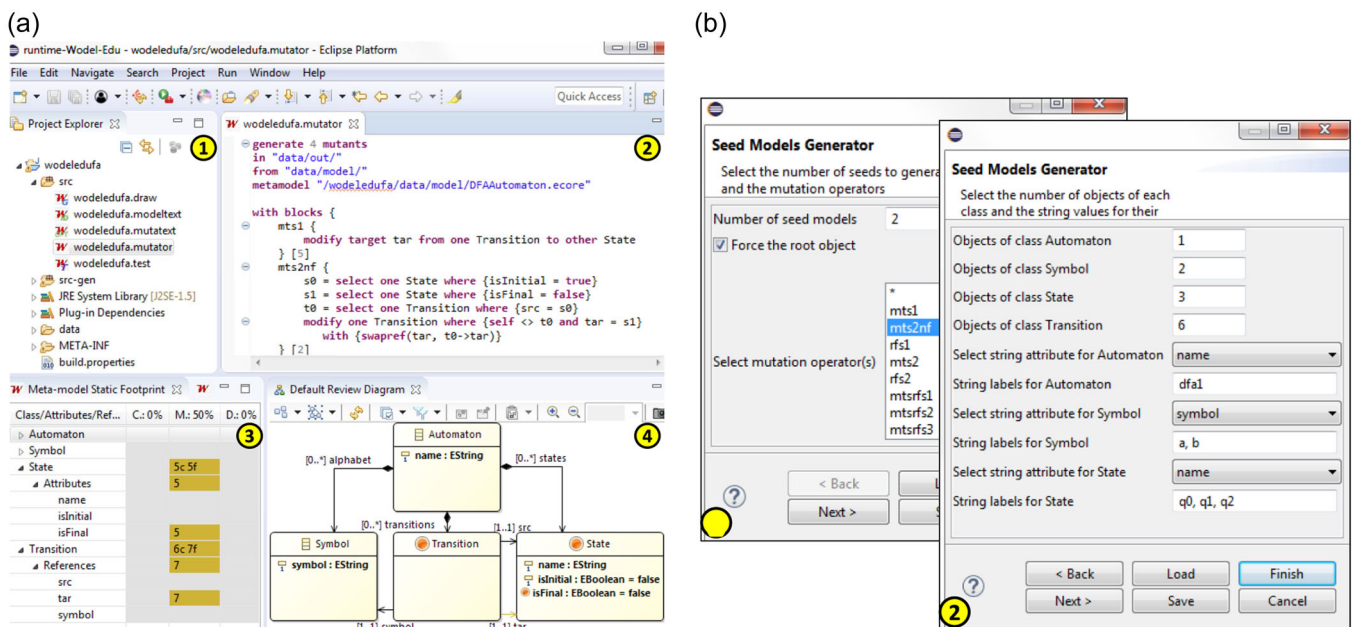


FIGURE 16 The WODEL-EDU environment (a) and first two pages of the seed model synthesizer wizard (b).

the answer to each RQ. We finish the section with a discussion of potential threats to the validity of the experiments in Section 7.4.

## 7.1 | RQ1 and RQ2: Quality of the generated exercises

In the following, we describe the experiment design to answer RQ1 and RQ2 (Section 7.1.1), report the experiment results (Section 7.1.2), and answer the RQs (Section 7.1.3).

### 7.1.1 | Experiment design

To answer RQ1 and RQ2, we designed a controlled experiment where students of a course on formal languages and automata theory, of the third year in the Computer Science degree at the Universidad Autónoma de Madrid, completed and ranked a set of exercises generated with WODEL-EDU.

Specifically, we generated a MOODLE questionnaire with 10 exercises about DFA, and a second questionnaire with 5 exercises for pushdown automata (PdA). The generated exercises are available at <http://moodle.wodel.eu> (user: demo; password: Wodel-Edu4Moodle).

A total of 68 students completed the first questionnaire, while 28 completed the second. The questionnaires were optional assignments of the course. To ensure a “best effort” in solving the exercises, and to promote student participation, we granted the participants up to 1.5 extra points (over 10) in the mark of the course.

After completing the exercises, students were asked to rank each kind of exercise between 1 (completely disagree) and 5 (completely agree) regarding the following statements:

- The exercise is easily understood.
- The exercise has an appropriate level of difficulty.
- The exercise is useful to learn automata.

To ensure a fair assessment, students could not see the exercise solutions until they had ranked these statements.

#### Finite Automata Questionnaire.

This questionnaire had two exercises of each of the five kinds supported. For its generation, we used the meta-model of the running example (Figure 2) and eight mutation operators.

The models used for the *match pairs* exercises were automatically created using the synthesizer described in

Section 5.2. We used a low complexity search configuration to generate the model of the first exercise (two alphabet symbols, two states, and four transitions), and an average complexity configuration to synthesize the model of the second exercise (two alphabet symbols, three states, and six transitions). For these two exercises, along with the three quality measurement statements, we asked the students which exercise was the most difficult to solve.

The students were given an hour to complete the questionnaire with only one attempt. Depending on their obtained grade, we granted the participants 0.2 up to 1.0 extra points in the final mark of the course.

#### Pushdown Automata Questionnaire.

We used the meta-model in Figure 17 to represent PdA. A PdA includes a set of States that have a name and can be initial and final; a set of inputAlphabet symbols accepted by the automaton; a set of stackAlphabet symbols used for the stack; a set of Actions that push a symbol into the stack or pop a symbol from it; and a set of Transitions between States. Transitions process a symbol of the input alphabet when their reference stackSymbol is the current symbol atop the stack, and then they perform the referred stack action.

The meta-model includes six OCL invariants. The invariants *one\_initial* and *some\_final* demand the existence of exactly one initial state and at least one final state. The invariants *inputAlphabet* and *stackAlphabet* ensure that the symbols of the input and stack alphabets are different from each other. Invariant *diff\_state\_name* demands that the states have different names. Finally, invariant *connected* requires that every state is accessible from the initial one.

We designed this second questionnaire taking into account that PdA are usually more complex than DFA. Hence, to keep a balance regarding the complexity between both assignments, we included five exercises in

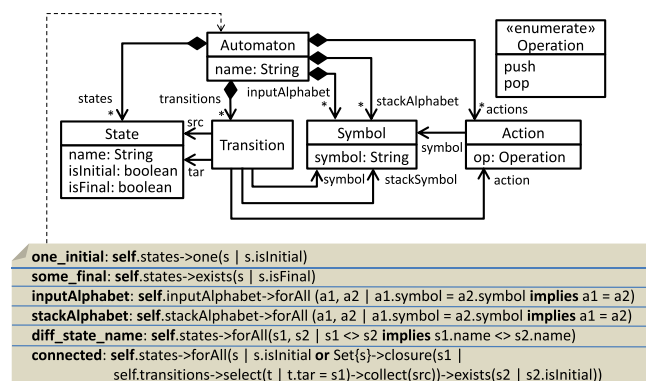


FIGURE 17 Meta-model for pushdown automata (PdA).

this second questionnaire (three of the *alternative response* kind and two of the *multiple diagram choice* kind) generated using one mutation operator.

We set 30 min with only one attempt to complete the questionnaire. As with the first questionnaire, students were asked to rate both kinds of exercises taking the same three measurements of quality. In this case, the extra points granted to the students were 0.1 up to 0.5, since this questionnaire included half number of exercises than the first one.

### 7.1.2 | Experiment results

A total of 68 students completed the questionnaire on DFA. Figure 18 shows the results of the evaluation for this set of exercises. The *alternative response* exercises obtained the highest ratings (89% easily understood, 90% appropriate level of difficulty, and 88% useful to learn automata). The *multiple diagram choice* exercises were identified as the hardest to understand (75%), as well as the most difficult ones (73%). This can be due to the inclusion of four different automata in the set of choices of this kind of exercises. In any case, both ratings were greater than 70%, which seems reasonable. Regarding the usefulness to learn automata, the *match pairs* exercises had the lowest rating (79%). This means that the usefulness of the five kinds of exercises was greater than approximately 80%, which overall is a very good rating. The average score of the students in this DFA questionnaire was 7.32 over 10.

Regarding the exercises created using the seed model synthesizer, 85% of the students identified the *match pairs* exercise generated with the high complexity configuration as the most difficult. These results show that there is a consensus that the higher the complexity

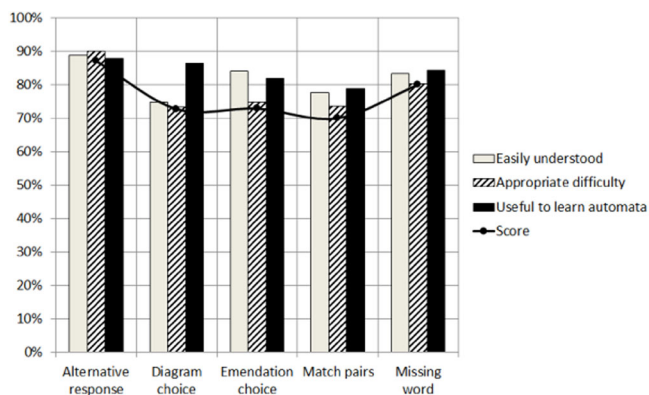


FIGURE 18 Results of the evaluation of the exercises for deterministic finite automata (DFA).

of the configuration used to synthesize the model, the more difficult the generated exercise is perceived, which is reasonable.

We calculated the correlations between the score of the exercises on DFA generated with WODEL-EDU, the scores of the partial and final exams, and the obtained rating on their difficulty. In the three cases, the correlations are near 0.5 (0.42, 0.44, and 0.48, respectively), which means there is a slight trend that the higher the score in the exercises generated with WODEL-EDU, the higher the scores in both exams, and the higher is the rating of the appropriateness regarding their difficulty.

The questionnaire on PdA was completed by 28 students of the same course. As we can observe in Figure 19, the average score of the *multiple diagram choice* exercises was lower than the average score of the *alternative response* exercises. On the contrary, the ratings regarding the three taken measurements show that the *multiple diagram choice* exercises were valued slightly better than the *alternative choice* ones. This can be due to the fact that the scores in the questionnaires were provided to the students after they had rated the exercises. The average score of the students in this questionnaire was 6.95 over 10.

The correlations between the score in this questionnaire on PdA, the scores of both exams, and the rating on the appropriateness of their difficulty, are 0.54, 0.51, and 0.21, respectively. This means there is a correlation between the score in the PdA exercises generated with WODEL-EDU and the scores in both exams, but there is no correlation with the rating of difficulty, as this value is close to 0. Note that the partial exam was taken before these MOODLE assignments.

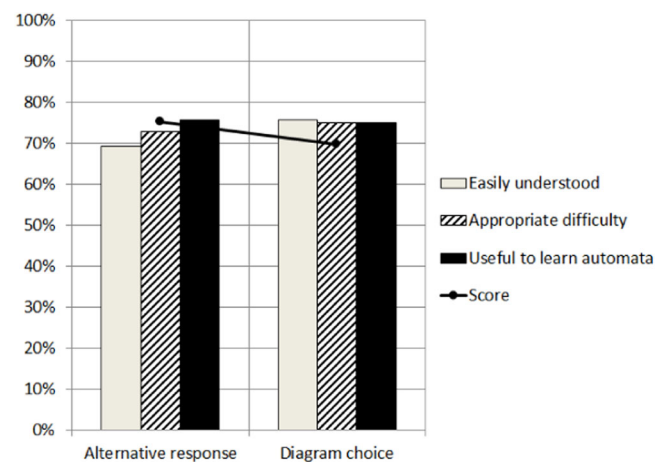


FIGURE 19 Results of the evaluation of the exercises for pushdown automata (PdA).

### 7.1.3 | Answering RQ1 and RQ2

Altogether, we can answer RQ1 and RQ2 affirmatively. Regarding RQ1, the ratings provided by the students to the statement “*The exercise is useful to learn automata*” in the questionnaire on DFA for the five kinds of exercises were greater than approximately 80%, with an average rating of 84%. In the case of the questionnaire on PdA, the lowest rating for this statement was 75%, and the average rating, including both questionnaires was 81%. Hence, we can state that the exercises generated with WODEL-EDU were found useful to learn automata.

Regarding RQ2, the lowest rating provided by the students to the statement ‘*The exercise is easily understood*’ was 73% for the *multiple diagram choice* kind in the questionnaire on DFA, and the average rating was 82%. The average rating for the ease of understanding in the questionnaire on PdA was 72%, and the average rating considering both questionnaires was 79%. Hence, these ratings allow us to conclude that the exercises generated with WODEL-EDU were found easy to understand.

## 7.2 | RQ3: Generality

Next, we describe the experiment designed to answer RQ3, and its results (Section 7.2.1), and answer RQ3 (Section 7.2.2).

### 7.2.1 | Experiment design and results

To answer RQ3, we have generated exercises for another domain: LCs. Figure 20 shows the meta-model for this notation. An LC consists of a set of Gates, which have a name, and hold a set of one or two InputPins and an OutputPin. An InputPin has a name and a reference src to an OutputPin, and an OutputPin has a name and a reference tar to an InputPin. A Gate can be of type AND, OR, or NOT.

The meta-model has OCL constraints requiring all the elements in the language to have different names (`diff_names`); the AND and ORGates to have two InputPins (`input_size_2`); and the NOTGates to have one InputPin (`input_size_1`).

Figure 21 shows an LC example conforming to this meta-model. The figure uses an abstract syntax at the top, and its standard concrete syntax at the bottom (with the Gates labeled with their name). This LC example has four initial InputPins (named ‘a’, ‘b’, ‘c’ and ‘d’), an input OR gate (‘or1’) with the InputPins ‘a’ and ‘b’, an input AND gate (‘and1’) with the InputPins ‘c’ and ‘d’, a NOT gate (‘not1’) with input the OutputPin

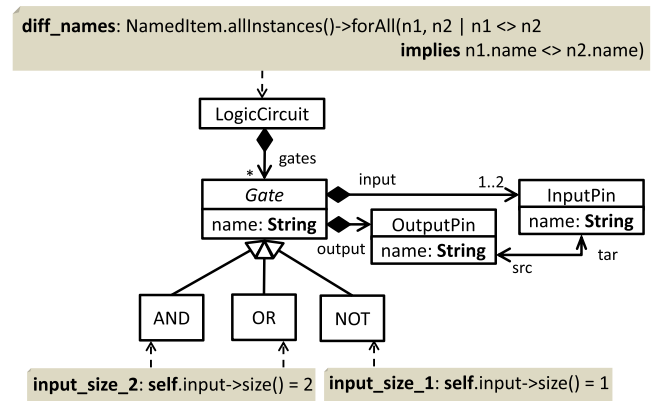


FIGURE 20 Meta-model for logic circuits (LCs).

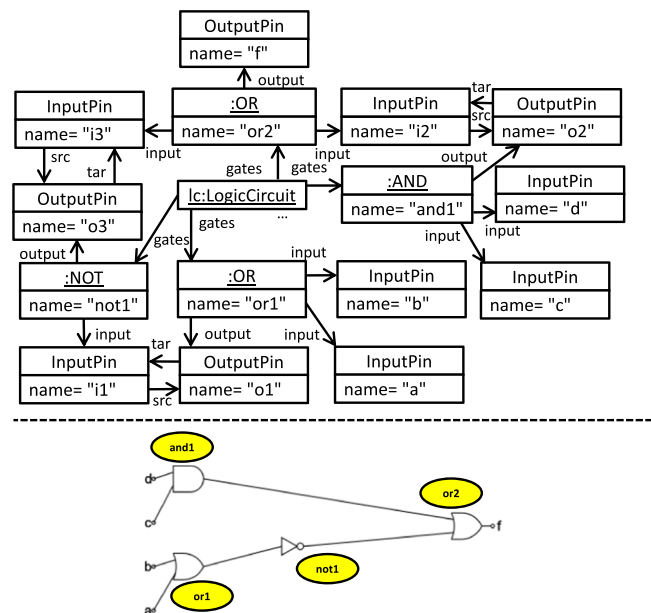


FIGURE 21 A valid logic circuit (LC) in abstract (top) and concrete (bottom) syntax.

of the input OR gate, and finally, an output OR gate (‘or2’) with inputs the OutputPins of the NOT and the AND gates, and output the OutputPin ‘f’.

Figure 22 illustrates the generation schema of an exercise of the kind *match pairs* for LCs. In this case, the alternative textual representations of the LCs are their corresponding boolean expressions.

The process starts with the seed model presented in Figure 21, which corresponds to  $\neg(a \vee b) \vee (c \wedge d)$  and applies to it a mutation operator that retypes an AND or an OR gate to an OR or an AND gate, respectively. The figure shows the resulting mutants in the middle with the mutated objects shaded in red. Please note that we implemented an equivalent detection algorithm to check if two LC models yield an equivalent boolean formula. This way, the resulting mutants are ensured to be nonequivalent.



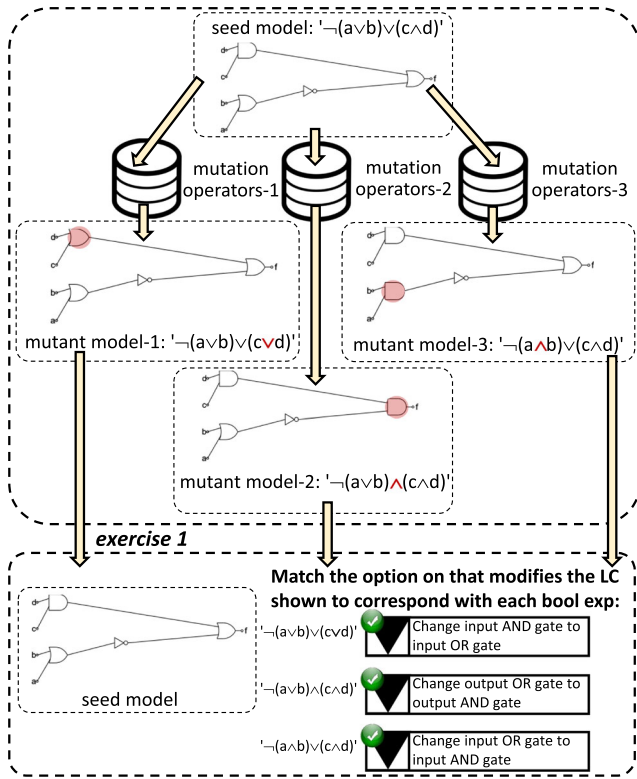


FIGURE 22 Schema of the match pairs choice exercises for logic circuits.

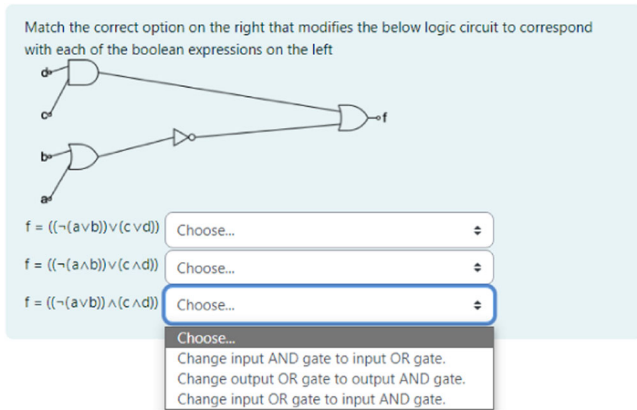


FIGURE 23 Generated match pairs choice exercises for logic circuits (MOODLE).

Finally, the bottom of the figure shows the generated exercise with the diagram corresponding to the seed model, and the description of each generated mutant using its corresponding boolean expression: ' $\neg(a \vee b) \vee (c \vee d)$ ', ' $\neg(a \vee b) \wedge (c \wedge d)$ ', and ' $\neg(a \wedge b) \vee (c \wedge d)$ '. The drop-down lists show the descriptions of the applied mutation operators in random order.

Figure 23 shows the resulting generated exercise for MOODLE. Using this strategy, we managed to synthesize LC exercises for each of the supported kinds, which are

TABLE 1 Specification sizes for the evaluation.

	DFA	PdA	LC
#Meta-model classes	4	5	8
#Mutation operators	8	1	8
#Seed models	10	5	6

Abbreviations: DFA, deterministic finite automata; LC, logic circuit; PdA, pushdown automata.

available at <http://moodle.wodel.eu> (user: demo; password: Wodel-Edu4Moodle).

### 7.2.2 | Answering RQ3

To check the generality of WODEL-EDU, we generated exercises for LCs. With this purpose, we used the extension points to generate alternative textual representations of the models to represent each LC with its corresponding boolean expression; and to detect equivalent mutants. The rest of the strategy to generate the different kinds of exercises supported follows the same pattern used in the example of DFA. Hence, we can state that WODEL-EDU is suitable to generate exercises for different domains, as long as the exercises are diagram-based, and models can be unambiguously asserted as being correct (the seed model) or incorrect (the mutants) solutions of an exercise statement.

## 7.3 | RQ4: Effectiveness and efficiency

To answer RQ4, we study the effectiveness of WODEL-EDU in generating exercises automatically, and the efficiency (time performance) of the process. Section 7.3.1 describes the design of the experiment, Section 7.3.2 reports the results, and Section 7.3.3 answers the RQ.

### 7.3.1 | Experiment design

To answer this RQ, we compare the specification size required to create exercises in each domain, and then measure the elements generated and the time taken.

Table 1 shows the specification sizes of the languages considered in the evaluation (DFA, PdA, and LCs), the number of mutation operators created, and the number of seed models manually created. Meta-model sizes range from 4 to 8 classes; for which we created between 1 and 8 mutation operators; and between 5 and 10 seed models. These numbers give an intuition of required manual effort.



**TABLE 2** Generation power.

	DFA	PdA	LC
#Synthesized seed models	2	–	–
#Mutants	344	24	98
Mutants per seed	28.7	4.8	16.4
#Alt. resp. exercises (1 mutant)	344	24	98
#Exercises per seed (1 mutant)	34.4	4.8	16.4
#Mult. choice exerc. (2 mutants)	172	12	49
#Exercises per seed (2 mutants)	17.2	2.4	8.2
#Match pairs exerc. (3 mutants)	114	8	32
#Exercises per seed (3 mutants)	11.4	1.6	5.4

Abbreviations: DFA, deterministic finite automata; LC, logic circuit; PdA, pushdown automata.

### 7.3.2 | Experiment results

Table 2 reports the elements generated automatically by WODEL-EDU. First, we only used the automated model synthesis in the case of DFA, where we generated two seed models with low and average complexity. Please note that we could have used this mechanism to generate the complete set of seed models, providing full automation for the process. Then, WODEL-EDU generated between 24 (for LC) and 344 (for DFA) mutants of the initial seed models. This means that we obtained between 4.8 and 28.7 mutants per seed model on average.

The following rows measure the number of exercises that are possible to generate out of the seed models and the mutants. This number depends on the mutants used per exercise kind. This way, the alternative response exercises use just a seed model and one mutant. This means that we can automatically generate between 24 and 344 different exercises, which gives an average of exercises per seed between 4.8 and 28.7. For exercises requiring two mutants—for example, multiple choice with three possible alternatives—we can generate between 12 and 172 different exercises automatically (without repeating mutants in options of different exercises), which gives between 2.4 and 17.2 exercises per initial seed model. Finally, for exercises requiring three mutants—like the match pairs—we can generate between 8 and 114 exercises, which results in 1.6–11.4 exercises per seed model.

Finally, Table 3 reports the generation time of the different elements. Regarding the seed model synthesis time, we only used this feature in the case of DFA, taking 2 s to generate them. The mutant generation time varied between 8 s (for PdA) and 1 min (for DFA). This yields a

**TABLE 3** Generation times per stage.

	DFA	PdA	LC	Total
Seeds	2 s.	–	–	2 s.
Mutants	1 m.	8 s.	24 s.	1 m. 32 s.
Resources	40 s.	7 s.	2 m. 30 s.	3 m. 17 s.
Moodle xml	3 s.	1 s.	3 s.	7 s.
Total	1 m. 45 s.	16 s.	2 m. 53 s.	4 m. 58 s.

Abbreviations: DFA, deterministic finite automata; LC, logic circuit; PdA, pushdown automata.

generation time of around 5 mutants per second. The resources generation time (e.g., images) was 40 s, 7 s, and 2 m. 30 s. for DFA, PdA, and LCs, respectively. We noticed that the technologies used to generate the LCs diagrams involve more processing work, thus this is reflected in the higher amount of time consumed in this example. However, we consider this time as still affordable.

Altogether, the total generation time was 4 m. 58 s. for the three examples, which seems overall efficient, taking into account that we have, as a result three questionnaires ready to be included in the selected MOODLE courses.

### 7.3.3 | Answering RQ4

As Table 2 shows, the generation power of WODEL-EDU is high: one seed model typically results in tens of – semantically distinct – mutants, which can be used to create tens of exercises. In the best case (DFA), out of 12 seed models (two generated automatically), WODEL-EDU created 344 mutants, from which we could create 344 alternative response exercises, 172 multiple choice exercises, and 114 match pairs exercises. In all the cases, there is at least one order of magnitude increase in the generation.

Regarding performance, the total generation time was 4 m. 58 s. for the three examples used in the evaluation: DFA, PdA, and LCs; with an average generation time of 1 m. 40 s. This seems a fast process to generate exercises in comparison to the time this task could have taken if we had to create these exercises by hand. Hence, we can conclude that WODEL-EDU is effective and efficient to generate exercises.

## 7.4 | Threats to validity

Next, we discuss the potential threats to validity associated with our evaluation.

### 7.4.1 | External validity

The external validity threat limits the ability to generalize the results beyond the experiment setting. For RQ1 and RQ2, our evaluation shows the usefulness and ease of understanding of the exercises generated by WODEL-EDU for the domain of automata. A more thorough evaluation of the exercises generated by WODEL-EDU on other domains would provide more conclusive results in this sense.

Regarding the questionnaire on PdA, it included half as many exercises as the questionnaire on DFA, and the participation in this part of the evaluation was not very high (28 students). Thus, we acknowledge that the weight of this part of the evaluation is not as high as the one for DFA.

We only included the automated generation of two seed models in the evaluation. A more exhaustive evaluation of the seed model synthesis and the measurement of the quality of the synthesized models comparing them with those created by hand would yield more weighted conclusions on this.

In the experiments for RQ1 and RQ2, the questions were evaluated from the point of view of the students. It would be interesting to perform an experiment to evaluate WODEL-EDU from the point of view of the professor.

For RQ3, we have studied the generality of the approach presenting another case study for LCs. We argue that WODEL-EDU can also be applied to other domains where solutions to problems can be expressed as diagrams—like electronic circuits or chemistry. However, it has limitations when a diagram cannot automatically be evaluated as a correct or incorrect solution to an exercise, like a UML class diagram with respect to requirements in natural language.

Similarly, for RQ4, we were limited to automata and LCs. The larger meta-model used in this evaluation is the LC meta-model, with 8 classes. Thus, we plan to conduct experiments in other domains with larger meta-models.

### 7.4.2 | Internal validity

This threat refers to scenarios where the evaluation results may indicate a causal relationship, although there is none. In our case, for RQ1 and RQ2, there might be a bias in the students who participated in the evaluation, since the questionnaires were optional, driving to be completed by those students who usually have a better performance. We reckon an evaluation of this kind would be strengthened if the exercises were provided as mandatory assignments. This would provide insight into the usefulness of the

exercises over the whole spectrum of students enrolled in the course. To alleviate this bias, we granted up to 1.5 points in the final mark of the course.

### 7.4.3 | Construct validity

This validity threat refers to the extent to which the experiment setting reflects the theory. We must point out that the evaluation of RQ1 and RQ2 was performed asynchronously without very thorough supervision. Most of the students completed the questionnaires from home, and we cannot assure you that they were not communicating with each other. This threat to validity could have been avoided if we had evaluated the students in person in a lab, though this was not possible due to the past pandemic situation. However, while this may bias the scores, it might not have an implication for the evaluation questions.

### 7.4.4 | Other threats

The mono-method bias refers to the fact that the experiments may under-represent reality. We have reduced this bias in RQ1 and RQ2 by taking three measurements in the analysis, along with the score obtained in the questionnaires. The threat of the hypothesis guessing reflects that the subjects may base their behaviors on their guesses. We reduced this threat by giving the students their scores in the questionnaires after they had given their evaluation ratings.

## 8 | CONCLUSIONS AND FUTURE WORK

This paper has presented WODEL-EDU, a mutation-based solution to automatically generate exercises for diagram-based languages. WODEL-EDU supports several mutation-based strategies to generate and correct five kinds of exercises automatically. We have assessed the usefulness and understandability of the generated exercises by the generation of two questionnaires for MOODLE on finite and pushdown automata, which we have used in a university course on automata theory. In addition, we have assessed the generality of the approach by generating exercises for LCs; and the effectiveness and efficiency of our solution by measuring the generation power and time performance. The results were good and prove the benefits of our approach.

We are currently working to improve WODEL-EDU to support gamification, and to generate other types of

exercises, e.g., generating options from the alternative textual representations, and supporting interactivity via direct diagram manipulation. Another line of future work is to generate exercises for other learning systems like edX or Coursera. We plan to synthesize questionnaires targeting mobile platforms, as well as random questionnaires with similar difficulty to create different questionnaires for each student. Another open line is to include adaptativity in questionnaires. Since mutations can model students' misconceptions, we could analyse the student wrong answers to generate specific exercises aiming to cover the detected weak points. In addition, we aim at using our approach in other domains, such as software design and electronic circuits. Finally, we plan to perform another evaluation of WODEL-EDU from the point of view of the professor.

## ACKNOWLEDGMENTS

We thank the reviewers for their useful comments. The study was supported by Spanish MICINN, Projects: PID2021-122270OB-I00 and TED2021-129381B-C21; R&D programme of the Madrid Region, Project: S2018/TCS-4314.

## DATA AVAILABILITY STATEMENT

The data that support the findings of this study are available from the corresponding author upon reasonable request.

## ORCID

Pablo Gómez-Abajo  <http://orcid.org/0000-0002-8319-4829>

Esther Guerra  <https://orcid.org/0000-0002-2818-2278>

Juan de Lara  <https://orcid.org/0000-0001-9425-6362>

## REFERENCES

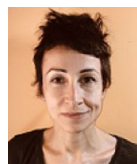
1. U. Z. Ahmed, M. Christakis, A. Efremov, N. Fernandez, A. Ghosh, A. Roychoudhury, and A. Singla, *Synthesizing tasks for block-based programming*, Advances in neural information processing systems 33: annual conference on neural information processing systems, NeurIPS (H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, eds.), 2020.
2. K. Ala-Mutka, *A survey of automated assessment approaches for programming assignments*, *comput. Sci. Educ.* **15** (2005), no. 2, 83–102.
3. R. Alur, L. D'Antoni, S. Gulwani, D. Kini, and M. Viswanathan, *Automated grading of DFA constructions*, AAAI Press, Washington, DC, 2013, pp. 1976–1982.
4. V. Aranega, J. Mottu, A. Etien, T. Degueule, B. Baudry, and J. Dekeyser, *Towards an automation of the mutation analysis dedicated to model transformation*, *STVR* **25** (2015), no. 5–7, 653–683.
5. W. Bian, O. Alam, and J. KienzeIs automated grading of models effective?: Assessing automated grading of class diagrams. *ACM MoDELS*, (2020), pp. 365–376.
6. M. Brambilla, J. Cabot, and M. Wimmer, *Model-driven software engineering in practice: Synthesis Lectures on software engineering*, 2nd ed., Morgan & Claypool Publishers, San Rafael, CA, 2017.
7. S. Foss, T. Urazova, and R. Lawrence, *Automatic generation and marking of UML database design diagrams*, SIGCSE 2022: The 53rd ACM Tech. Sympos. Comput. Sci. Educ. (L. Merkle, M. Doyle, J. Sheard, L. Soh, and B. Dorn eds.), vol. **1**, ACM, 2022, pp. 626–632.
8. P. Gómez-Abajo, E. Guerra, and J. de Lara, *A domain-specific language for model mutation and its application to the automated generation of exercises*, *Comput. Lang. Syst. Struct.* **49** (2017), 152–173.
9. P. Gómez-Abajo, E. Guerra, de J. Lara, and M. Merayo, *Seed model synthesis for testing model-based mutation operators*, Advanced information systems engineering, Springer International Publishing, New York, 2020, pp. 64–76.
10. P. Gómez-Abajo, E. Guerra, J. de Lara, and M. Merayo, *Systematic engineering of mutation operators*, *J. Obj. Technol.* **19** (2020), no. 3, 3:1–16; Special Issue dedicated to Martin Gogolla on his 65th Birthday.
11. P. Gómez-Abajo, E. Guerra, J. de Lara, and M. G. Merayo, *A tool for domain-independent model mutation*, *Sci. Comput. Program.* **163** (2018), 85–92.
12. L. Gong, *Auto-grading dynamic programming language assignments*, University of California, Berkeley, Tech. Rep., 2014.
13. E. Guerra, J. SánchezCuadrado, and J. de Lara, *Towards effective mutation testing for ATL*, *MoDELS*, IEEE, 2019, pp. 78–88.
14. G. Hoggarth and M. A. Lockyer, *An automated student diagram assessment system*, *ITiCSE*, ACM, 1998, pp. 122–124.
15. D. Jackson, *Alloy: a language and tool for exploring software designs*, *Commun. ACM.* **62** (2019), no. 9, 66–76.
16. C. Kotsiopoulos, I. Doudoumis, P. Raftopoulou, and C. Tryfonopoulos, *DaST: an online platform for automated exercise generation and solving in the data science domain*, *CSERC*, ACM, 2019, pp. 104–109.
17. M. Kuhlmann, and M. Gogolla, *From UML and OCL to relational logic and back*, *MoDELS LNCS*, **7590**, Springer, New York, 2012, pp. 415–431.
18. X. Liu, S. Wang, P. Wang, and D. Wu, *Automatic grading of programming assignments: an approach based on formal semantics*, *ICSE-SEET*, 2019, pp. 126–137.
19. R. Lobb and J. Harlow, *Coderunner: a tool for assessing computer programming skills*, *ACM Inroads.* **7** (2016), no. 1, 47–51.
20. J. McBroom, I. Koprinska, and K. Yacef, *A survey of automated programming hint generation: the HINTS framework*, *ACM Comput. Surv.* **54** (2022), no. 8, 172:1–172:27.
21. M. Mernik, J. Heering, and A. M. Sloane, *When and how to develop domain-specific languages*, *ACM Comput. Surv.* **37** (2005), no. 4, 316–344.
22. D. S. Mishra and S. H. Edwards, *The programming exercise markup language: Towards reducing the effort needed to use automated grading tools*, Proceedings of the 54th ACM technical symposium on computer science education, SIGCSE 2023 (M. Doyle, B. Stephenson, B. Dorn, L. Soh, and L. Battestilli, eds.), vol. **1**, ACM, 2023, pp. 395–401.
23. Moodle project, Moodle website, 2023. <https://moodle.org/>
24. Object Management Group, UML 2.4 OCL specification, 2014. <http://www.omg.org/spec/OCL/>

25. A. Papasalouros, *Automatic exercise generation in euclidean geometry*, Artificial intelligence applications and innovations, Springer, Berlin Heidelberg, 2013, pp. 141–150.
26. V.-A. Pădurean, G. Tzannetos, and A. Singla, *Neural task synthesis for visual programming*, arXiv (2305.18342), cs.LG, 2023.
27. D. Sadigh, S. A. Seshia, and M. Gupta, *Automating exercise generation: A step towards meeting the MOOC challenge for embedded systems*, WESE, ACM, 2013, pp. 2:1–2:8.
28. J. SánchezCuadrado and M. Gogolla, *Model finding in the EMF ecosystem*, J. Object Technol. **19** (2020), no. 2, 10:1–21.
29. M. Snoeck, R. Haesen, H. Buelens, M. D. backer, and G. monsieur, *computer aided modelling exercises*, Informatics Educ. **6** (2007), no. 1, 231–248.
30. J. Soler, I. Boada, F. Prados, J. Poch, and R. Fabregat, *A web-based e-learning tool for uml class diagrams*, EDUCON, 2010, pp. 973–979.
31. E. Stankov, M. Jovanov, and A. M. Bogdanova, *Smart generation of code tracing questions for assessment in introductory programming*, Comput. Appl. Eng. Educ. **31** (2023), no. 1, 5–25.
32. F. Steiner, B. Lueger, B. Wallisch, and T. Polzer, *Automated evaluation system for microcontroller assignments*, IEEE Global Eng. Educ. Conf., EDUCON 2023, IEEE, 2023, pp. 1–7.
33. C. M. Tang, Y. Yu, and C. K. Poon, *An automated system with a versatile test oracle for assessing student programs*, Comput. Appl. Eng. Educ. **31** (2023), no. 1, 176–199.
34. P. Thomas, K. Waugh, and N. Smith, *Experiments in the automatic marking of ER-diagrams*, SIGCSE Bull. **37** (2005), no. 3, 158–162.
35. P. Thomas, K. Waugh, and N. Smith, *Generalised diagram revision tools with automatic marking*, SIGCSE Bull. **41** (2009), no. 3, 318–322.
36. K. Wang, R. Singh, and Z. Su, *Search, align, and repair: Data-driven feedback generation for introductory programming exercises*, PLDI, Association for Computing Machinery, New York, 2018, pp. 481–495.
37. T.-L. Wong, C. K. Poon, C. M. Tang, Y. T. Yu, and V. C. S. Lee, *Automatic generation of matching rules for programming exercise assessment*, Technology in education. Innovations for online teaching and learning, Springer Singapore, 2020, pp. 126–135.
38. J. Xia and C. B. Zilles, *Using context-free grammars to scaffold and automate feedback in precise mathematical writing*, Proc. 54th ACM Tech. Sympos. Comput. Sci. Educ., SIGCSE 2023 (M. Doyle, B. Stephenson, B. Dorn, L. Soh, and L. Battestilli, eds.), vol. **1**, ACM, 2023, pp. 479–485.

## AUTHOR BIOGRAPHIES



**Pablo Gómez-Abajo** is an assistant professor at the computer science department of the Universidad Autónoma de Madrid. After around 8 years of working in the industry, he returned to academia and joined the modeling and software engineering research group in 2015. Contact him at [pablo.gomez@uam.es](mailto:pablo.gomez@uam.es).



**Esther Guerra** is an associate professor at the computer science department of the Universidad Autónoma de Madrid. Together with J. de Lara, she leads the modeling and software engineering research group (<http://miso.es>). She is interested in flexible modeling, meta-modeling, domain-specific languages, and model transformation. Contact her at [esther.guerra@uam.es](mailto:esther.guerra@uam.es).



**Juan de Lara** is full professor at the computer science department of the Universidad Autónoma de Madrid. Together with E. Guerra, he leads the modeling and software engineering research group. His research interests are in model-driven engineering and automated software development. Contact him at [juan.delara@uam.es](mailto:juan.delara@uam.es).

**How to cite this article:** P. Gómez-Abajo, E. Guerra, and J. de Lara, *Automated generation and correction of diagram-based exercises for Moodle*, Comput. Appl. Eng. Educ. (2023), 1–22. <https://doi.org/10.1002/cae.22676>