A language-parametric test amplification framework for executable domain-specific languages

Faezeh Khorram, Erwan Bousse, Jean-Marie Mottu, Gerson Sunyé, Djamel Eddine Khelladi, Pablo Gómez-Abajo, Pablo Cañizares, Esther Guerra, Juan de Lara

A Language-Parametric Test Amplification Framework for Executable Domain-Specific Languages

Faezeh Khorram[1*], Erwan Bousse[2], Jean-Marie Mottu[2], Gerson Sunyé[2], Djamel Eddine Khelladi[1], Pablo Gómez-Abajo[3], Pablo C. Cañizares[4], Esther Guerra[3], Juan de Lara[3]

[1*]CNRS, University of Rennes, Rennes, France.
[2]Nantes Université, IMT Atlantique, CNRS, LS2N, UMR 6004, Nantes, France.
[3]Universidad Autónoma de Madrid, Madrid, Spain.
[4]Universidad Complutense de Madrid, Madrid, Spain.

*Corresponding author(s). E-mail(s): faezeh.khorram@inria.fr;
Contributing authors: erwan.bousse@ls2n.fr; jean-marie.mottu@ls2n.fr; gerson.sunye@ls2n.fr; djamel-eddine.khelladi@irisa.fr; pablo.gomeza@uam.es; pablocc@ucm.es; esther.guerra@uam.es; juan.delara@uam.es;

Abstract

Behavioral models are important assets that must be thoroughly verified early in the design process. This can be achieved with manually-written test cases that embed carefully hand-picked domain-specific input data. However, such test cases may not always reach the desired level of quality, such as high coverage or being able to localize faults efficiently. *Test amplification* is an interesting emergent approach to improve a test suite by automatically generating new test cases out of existing manually-written ones. Yet, while ad-hoc test amplification solutions have been proposed for a few programming languages, no solution currently exists for amplifying the test suites of behavioral models.

In order to fill this gap, we propose an automated and generic test amplification approach for executable Domain Specific Languages (DSLs). Hence, given an executable DSL, a conforming behavioral model, and an existing test suite, our approach synthesizes new regression test cases in three steps: (i) generating new test inputs by applying a set of generic modifiers on the existing test inputs; (ii) running the model under test with new inputs and generating assertions from the execution traces; and (iii) selecting the new test cases that increase the initial test quality. We provide a textual DSL to control and configure the amplification process, along with tool support for the whole approach atop the Eclipse GEMOC Studio. For assessment, we report on empirical evaluations over two different executable DSLs, which show improved test quality in terms of both coverage and mutation score.

Keywords: Test Amplification, Regression Testing, Mutation Testing, Executable Model, Executable DSL

1 Introduction

Many Domain-Specific Languages (DSLs) are used for describing the dynamic behavior of systems as *behavioral models* (e. g., state machines [1], activity diagrams [2], and process models [3, 4]). They are used in dedicated environments with tool support that includes dynamic verification and validation (V&V) techniques, enabling the user (i. e., the

domain expert) to assess the correctness of the modeled behavior as early as possible [5]. Dynamic V&V techniques require the execution of the models, hence their application is restricted to DSLs with translational or operational semantics, i. e., compilation or interpretation, respectively. We focus on DSLs with operational semantics, referred to as *executable DSLs (xDSLs)*.

Testing is a popular dynamic V&V technique that involves executing systems and observing whether they act as expected. Currently, several testing approaches are proposed for xDSLs, some tailored for specific ones [6–9], while some others provide generic solutions that are applicable to a wide range of xDSLs [10–14]. They allow domain experts to write and execute test cases for behavioral models. However, writing test cases with a high level of quality (e. g., having high coverage or being able to localize faults in the model efficiently) is a challenging manual task [15].

In the realm of software testing, a test case generation technique called *test amplification* has recently emerged [16]. This technique is able to improve an existing manually-written test suite by generating new test cases towards a specific goal (e. g., improve coverage or increase the accuracy of fault localization). In a nutshell, a test amplifier creates slight variations in existing test cases in order to put the system in unexplored states, and then generates new oracles in a way adapted to the chosen goal — e. g., oracles directly based on the execution traces to strengthen regression testing [16, 17].

So far, several test amplification solutions have been proposed for specific programming languages, grounded in their supporting testing frameworks (e. g., Java [18], Pharo Smalltalk [19], and Python [20]). Bringing test amplification to xDSLs could greatly help domain experts to create satisfying test suites for their behavioral models. Yet, in a context where there are plenty of xDSLs to define behavioral models [1–4] and the engineering of new ones is recurrent [21], developing a test amplification approach for each and every xDSL is costly and potentially repetitive.

In this paper, we propose a generic, automated approach for amplifying test suites of executable models. We focus on regression testing as a goal for test case generation and use as a starting point our earlier testing framework presented in [13], which supports the definition and execution of test cases for any executable model using the standard Test Description Language (TDL) [22]. Given an xDSL, a conforming behavioral model, and a TDL test suite for this model, our proposed approach amplifies this test suite in three steps. First, new test input data is generated by applying a set of modifiers to the input data of existing test cases. For primitive data, we adapt modifiers suggested for JUnit test cases [18, 23], and for more complex data related to the modeling nature of the tested system, we propose a set of new modifiers. The output of this phase is a set of new test cases but without any assertions. Second, each newly generated test case is executed, and from the resulting trace, all possible assertions for the new test case are generated. Third, those new test cases improving the quality of the original test case are selected, hence only the most efficient ones are proposed to the domain expert.

For measuring test quality, we support both model element coverage [24] and mutation analysis [25, 26] metrics. Applying mutation analysis relies on having a set of mutation operators for the DSL that the tested model conforms to. To better support DSLs for which no mutation operators exist off the shelf, the proposed framework is able to automatically generate a set of mutation operators for any given DSL. Lastly, we provide a textual DSL that can be used to control and configure all aspects of the amplification process.

We implemented the proposed framework for the Eclipse GEMOC Studio [27], a language and modeling workbench for xDSLs. We ran two experiments with 71 test suites written for 71 models conforming to two different xDSLs. We successfully generated (i) 32 new test cases improving model element coverage ranging from 6.19 % to 34.62 % (first experiment); and (ii) 209 new test cases improving mutation score ranging from 4.06 % to 55.10 % (second experiment). Altogether, these results demonstrate the effectiveness of test amplification in the context of executable model testing. We also evaluated the efficiency of the automatic mutation operator generator and we observed it can both emulate the manually-defined mutation operators and also generate operators with a high percentage of metamodel coverage.

This paper is an extension of our MODELS'22 paper [28]. New contents include supporting model element coverage for test case selection, automated generation of mutation operators, the textual DSL for the configuration of the amplification process, an extended evaluation that considers element coverage in addition to mutation score as test case selection criterion, and an additional experiment that evaluates

the effectiveness of the new facility for automated mutation operator generation.

Paper organization. Section 2 provides the background, a running example, and the motivation of our proposal. The proposed approach is then introduced in Sections 3 to 5, and its supporting tool is explained in Section 6. Section 7 presents the evaluation of our approach. Finally, the related work is provided in Section 8, and Section 9 concludes the paper.

2 Background

This section introduces a running example that will be used across the paper (Section 2.1) and provides the required background concepts (Sections 2.2- 2.5).

2.1 Running Example: Arduino

Arduino[1] is an open-source company that offers hardware boards with embedded CPUs, and different modules (e. g., sensors, LEDs, actuators) that can be attached to a board. An Integrated Development Environment (IDE) is available to develop programs (called sketches) for such boards in C or C++. We consider as a running example a sample executable DSL aiming at easing the modeling and early simulation of Arduino boards along their sketches. Figure 1 shows an excerpt of the definition of such Arduino xDSL[2], which is further introduced in the following sections. From now on, we refer to this xDSL as xArduino.

2.2 Executable DSL (xDSL)

In this paper, we target xDSLs that are composed of at least (i) an abstract syntax specifying the domain concepts; (ii) an operational semantics enabling the execution of the models conforming to the xDSL [27]; and (iii) a behavioral interface defining how to interact with a running model [29].

2.2.1 Abstract Syntax

We consider the abstract syntax of an xDSL to be defined by an Ecore metamodel [30]. This is a set of metaclasses, each containing a set of features, i. e., either an attribute with primitive type or a reference to another metaclass. Figure 1(**a**) shows the abstract syntax of xArduino. An xArduino model starts with a root Project element that may contain several Board

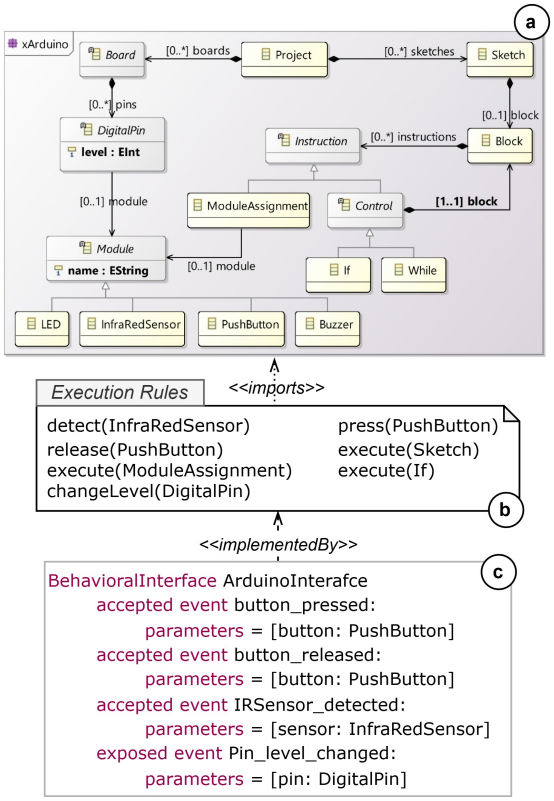


Fig. 1 An excerpt of an Arduino xDSL (called xArduino)

and Sketch elements. A Board represents a physical board, containing several DigitalPins, each one of them associated with a Module such as LED, InfraRedSensor, PushButton, and Buzzer. DigitalPin has a level integer attribute, which represents the state of its Module. For example, when the level for a DigitalPin connected to a PushButton is equal to 1, it means the button is being pressed. Sketch elements represent the board's behavior. They may contain a Block with Instructions such as ModuleAssignment for changing the state of a Module, and Control instructions for conditional behaviors (e. g., using If or While).

Figure 2 shows an example xArduino model depicted using a graphical concrete syntax. Four Module instances including a PushButton, an InfraRedSensor, a white LED, and a Buzzer are connected to different DigitalPins of an Arduino Board (on the top). The board's behavior, modeled on the bottom of Figure 2 using a Sketch element, is: "`if button1` is pressed, the `white LED` turns on (i. e., activating the system) and then if the `infrared sensor` detects an obstacle, the `buzzer` alternates between noise/silence periods twice (i. e., reporting an intrusion). Otherwise, the `white LED` turns off".

[1] https://www.arduino.cc/
[2] Inspired from https://github.com/mbats/arduino

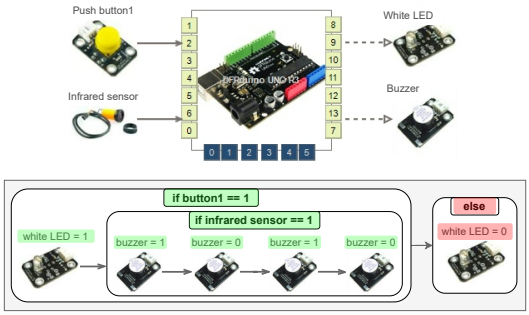


Fig. 2 An example xArduino model. The covered and not-covered elements by the test case of Figure 3 are highlighted in green and red, respectively

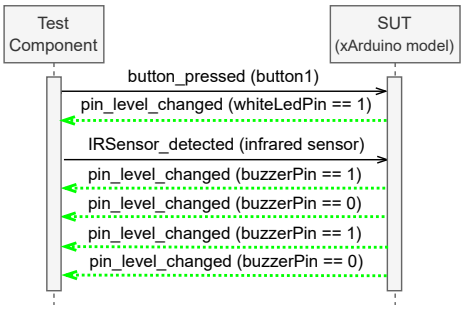


Fig. 3 A TDL test case for the xArduino model of Figure 2 (dotted arrows are passed assertions)

2.2.2 Operational Semantics

The operational semantics of an xDSL defines how to execute a model conforming to the xDSL's abstract syntax. It should include two parts: the definition of the possible runtime states of a running model, and a set of execution rules that change such runtime state over time.

We assume that the runtime state is defined in the metamodel in the form of additional dynamic attributes and references. For example, in Figure 1(**a**), the `level` attribute of the DigitalPin, shown in bold, represents the runtime state of its associated Module since changing its value at runtime puts an xArduino model in different states. Next, for each metaclass of the abstract syntax that has a runtime behavior, an execution rule is needed to implement such behavior and the execution rules may call each other to complement the model execution. Figure 1(**b**) lists an excerpt of the xArduino execution rules. For example, the `detect` rule implements the behavior of detecting an obstacle by an InfraRedSensor. Note that this paper only considers xDSLs with discrete-event operational semantics (i. e., not continuous) and with deterministic behavior. This guarantees that for the same input, the execution result is always the same.

2.2.3 Behavioral Interface

The behavioral interface of an xDSL defines how a conforming model can interact with its environment through *events*, and must be implemented by the execution rules of the operational semantics [29]. Figure 1(**c**) presents a behavioral interface for xArduino. It comprises a set of events, each containing parameters conforming to the xDSL's abstract syntax. Considering a running model, accepted events indicate the kinds of requests that the model accepts, and exposed events determine the observable reactions of the model. This means the execution trace of an executable model can be defined as a sequence of exposed event instances. For example, when executing an xArduino model, it is possible to simulate the pressing and releasing of a button (using `button_pressed` and `button_released` events, respectively), and the detection of an obstacle by a sensor (using `IRSensor_detected` event). Whenever the `level` of a DigitalPin changes, it will be exposed by instantiating the `Pin_level_changed` event for the related DigitalPin. This paper assumes the accepted events are processed following a run-to-completion semantics.

2.3 Testing Support for xDSLs

An xDSL with testing support enables domain experts to test conforming behavioral models early in the design phase. In [13], we proposed a generic testing framework for xDSLs using the standard Test Description Language (TDL) [22]. Given an xDSL, the framework generates an xDSL-specific TDL library that provides facilities to define test cases for the conforming models based on the xDSL definition, and then executing them and obtaining the test verdict (i. e., pass, fail). It also supports xDSLs with behavioral interfaces, meaning that it is possible to write event-driven test cases in which the test case interacts with a running model under test based on what the behavioral interface offers. In such test cases, the test input data and expected output (i. e., used in the assertions, defining the test oracle) are defined as a sequence of accepted event and exposed event instances, respectively.

For example, Figure 3 shows a TDL test case for the xArduino model from Figure 2. The test case is defined as a scenario of exchanging messages between the *test component*—i. e., the test executor—and the *System Under Test* (SUT)—in our case the xArduino model. When the test component is the sender, the exchanged message carries test input data, i. e., an accepted event instantiated from the behavioral interface of the xArduino DSL. When the test component is the receiver, the messages from the SUT are the expected output (i. e., the exposed event instances), and their carried data are checked by assertions defining the test oracle. The values of the events' parameters are elements of the xArduino model with values of their runtime features (such as `whiteLedPin` with `level == 1`). This test case validates whether the `white LED` turns on when `button1` is pressed, and then whether the `buzzer` alternates between noise/silence periods when the `Infrared sensor` detects an obstacle. Therefore, the test case is checking the Sketch shown in the bottom part of the xArduino model from Figure 2, except for the `else` part. As shown by the dotted arrows highlighted in green in Figure 3, the test case is a success.

In previous work [13], we have considered how to design test cases (test data and oracle) and run them automatically to get a verdict. This allows the domain expert to define test cases, but so far manually (cf. top-right corner of Figure 7, which will be described in the next section). In this paper, we go further by providing domain experts with assistance in defining new test cases by amplification (cf. the rest of Figure 7).

When providing test support for xDSLs, we promote Model-driven Engineering (MDE) to design the xDSLs, their conforming models, and finally their tests. Therefore, we focus on testing the executable models directly. They can already be run on the test cases without considering the generated code (which may be deployed, but probably already too late for early testing). We thus help the tester to fight against the modeling errors the domain expert may have introduced into the model.

2.4 Test Quality Evaluation

Once test cases are defined for a model, it is important to measure their quality to identify whether they are good enough or an improvement is needed. In the literature, there are in particular two popular test quality measurement techniques: coverage computation and mutation analysis. We review both of them next.

2.4.1 Coverage Computation

Test coverage measures how much of a SUT is executed by a given test case based on a given criterion [31]. In the context of xDSLs, where models are the SUT, we previously proposed a generic coverage metric named *element coverage* [24]. It computes the percentage of the covered model elements by a given test case using two ingredients: the definition of the conforming xDSL, and the model execution traces. By analyzing the xDSL definition, we identify those types of elements whose execution can be captured in a trace, called "traceable" elements. Then, given an execution trace, the captured model elements are considered as *covered*, the not-captured "traceable" elements as *not-covered*, and the rest have *no coverage status*.

For example, we computed the element coverage of the running example—the coverage of the xArduino model of Figure 2 by the test case of Figure 3—and the result is shown in Figure 2 in the form of elements highlighted in green if covered, and in red if not covered. As can be seen, the test case does not cover the else part of the Sketch.

2.4.2 Mutation Analysis

Mutation analysis is a technique that injects artificial faults in the SUT, and then measures the degree to which an existing test case detects them [25, 26]. Artificial faults are modeled via *mutation operators*, which perform small modifications (e. g., flipping '>' by '<' in an expression, changing the value of constants) on the source code, or on the model when considering xDSL testing. These operators are systematically applied to the SUT to produce sets of *mutants*. For example, Figure 4 shows part of a mutant for the xArduino model of Figure 2, generated by changing a constant within the `else` part.

Mutants can be used to evaluate the quality of the test cases. If a test case distinguishes the original program from the mutant—i. e., some oracle fails on the mutant, but none on the original—then the test case has detected the fault, and we say that the mutant has been killed. For example, if we run the test case of Figure 3 on the mutant of Figure 4, the assertions evaluate just like on the original model of Figure 2. Since the mutation is performed on the `else` part, which is not covered by the test case, it is not able to kill the mutant. The mutation score of a test suite—the percentage of mutants killed by its test cases—provides a

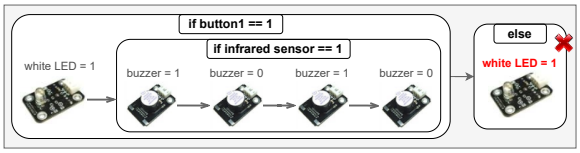


Fig. 4 Part of a mutant generated from the xArduino model of Figure 2 by changing the assignment of the `else` part

measure of its quality [32], and can be used as criteria to extend a test suite with more test cases.

In the context of xDSLs, we previously proposed WODEL, a generic model mutation framework which (i) provides a DSL for the definition of mutation operators for a given xDSL; (ii) automatically generates mutants from the conforming models by the application of operators; and (iii) computes the mutation score of the models' test suites [33].

2.5 Test Amplification

Test amplification refers to all the existing techniques aiming at enhancing manually-written test cases based on a specific goal, such as improving the coverage of changes or increasing the accuracy of fault localization [16]. A subset of these techniques is focused on improving manually-written test cases to avoid regressions [17, 34]. Given a test suite for a system, such techniques create new test cases by modifying the test input data of its test cases and then run the system with this modified data to put the system in unexplored states. For each new test case, an oracle is generated by inferring assertions from the resulting execution trace of the system. As new test cases are based on the current behavior of the system, these techniques can effectively strengthen regression testing [17, 18]. Since test amplification may generate large amounts of test cases, it is important to keep only the relevant ones. The most used techniques for identifying relevant test cases are coverage computation and mutation analysis [16] (cf. Section 2.4).

For instance, let us consider the Java source code of method factorial in Figure 5. A tester may have implemented a first test case named testfactorial(), shown in lines 1–4 of Figure 6. Amplifying this test case may generate the new test case testfactorialAmplified(), shown in lines 6–9 of Figure 6, where the test data has been changed to 0. Running the new test case on the method permits knowing that the output is expected to be 1, which is used to generate the test oracle. Finally, the new test case is selected since it

```
1  public static int factorial(int n) {
2      if (n == 0)
3          return 1;
4      else
5          return n * factorial(n-1);
6  }
7 }
```

Fig. 5 Java source code of factorial (without managing exception)

```
1  @Test
2  void testfactorial() {
3      assertEquals(24, factorial(4));
4  }
5
6  @Test
7  void testfactorialAmplified() {
8      assertEquals(1, factorial(0));
9  }
```

Fig. 6 A JUnit test case of method factorial and an amplified one

covers the 3rd line of the method, which the first test case did not cover.

3 Objectives and Approach Overview

In this section, we first provide the motivation and the objectives of this paper (Section 3.1). Then, we present an overview of our proposed test amplification framework (Section 3.2).

3.1 Motivation and Objectives

As described above, amplifying tests for improving regression testing involves manipulating test data. Indeed, test amplification consists of modifying test input data in conformance with the SUT and generating new assertions from the execution traces. Both tasks require working with system-specific data, which may take very different shapes depending on the language the system is implemented with. Hence, existing test amplification solutions are dedicated to specific programming languages and are grounded on their supporting testing frameworks (e. g., Java [18], Pharo Smalltalk [19], and Python [20]).

In the context of this paper, the SUT is an executable model defined by an xDSL. As there are a variety of xDSLs in different domains [1–4] and engineering of new ones is recurrent [21], providing test amplification for every one of them represents a very time-consuming, error-prone, and repetitive task. A promising solution would be a *generic* test amplification approach applicable to xDSLs which must be founded on a generic testing framework for xDSLs that supports test quality measurement. As our earlier

testing framework [13] meets these requirements, this paper uses it as the basis for proposing a generic test amplification framework for xDSLs.

Considered objectives

As mentioned above, test amplification involves modifying test cases of a given model, and a modification is valid and meaningful if it relies on the specification of the model's conforming xDSL. Accordingly, we aim for the following objective:

> **Objective 1:** A language-parametric *test amplification* approach for xDSLs which uses an xDSL's definition to amplify the test cases of its conforming models.

While we can step forward to a generic test amplification approach, different strategies can be followed when running a test amplification process. For example, the new test cases generated by amplification must be evaluated to measure the level of achieved improvement. This evaluation can be performed based on different criteria such as coverage computation and mutation analysis, each with particular advantages and disadvantages. A more comprehensive approach should support various metrics and allow the language engineers to set up the process based on their preferences. Thus, we also aim for the following objective:

> **Objective 2:** Enabling language engineers to set up different test amplification strategies for a given xDSL (e. g., applying various metrics for measuring the obtained test improvement).

Despite the benefits of using mutation analysis for the test improvement measurement, it adds up to the tasks of the language engineers since the technique needs a set of mutation operators to generate mutants from a model under test. In the literature, various facilities are proposed so far aiming at easing this task for the language engineers (such as the WODEL framework [33]) but the definition of specific mutation operators for particular DSLs is still a manual endeavor. However, since the model mutation operators usually perform certain *types* of mutations, it is apparent that we can generalize this task and reduce this manual effort. Accordingly, we defined our last objective as follows:

> **Objective 3:** An automatic mutation operator generator for xDSLs that can be configured by the language engineers.

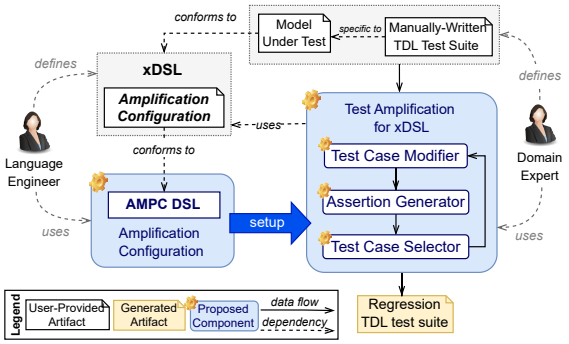


Fig. 7 Overview of the approach

In the next section, we propose a generic test amplification approach for xDSLs that addresses these objectives.

3.2 Approach Overview

Figure 7 shows an overview of the proposed approach. We consider that a language engineer (on the left) defines an xDSL according to the definitions given in Section 2.2, to be used by a domain expert (on the right) for defining models. The domain expert also provides TDL test suites for the models to verify their behavior.

We specialized the existing test amplification approach targeting conventional software (described in Section 2.5) for the context of models, realized by the *Test Amplification* component (on the bottom right).

This component embeds three connected parts, altogether generating an improved version of the given test suite for regression testing of the model, as follows. First, the *Test Case Modifier* generates new test cases by performing a set of modifications on the input data of the test cases of the input test suite. Then, the *Assertion Generator* uses the model execution traces achieved by running the new test cases on the model, to complete the definition of the new test cases with regression assertions. Finally, the *Test Case Selector* filters the generated test cases based on a given metric (coverage, mutation score, or a combination of both) to keep only the most effective ones. These steps can be iterated until a stop criterion is met (e. g., after a specific number of iterations). Section 4 will provide detailed information.

We enable the language engineers to customize the *Test Amplification* component for their xDSLs by defining an *Amplification Configuration* model using a

dedicated textual DSL named **AMPC** (on the bottom left). The AMPC language allows specifying the kind of modifications to be applied to the input test suite, their application policy, the test selection metric to use for filtering, and the stop criterion for the amplification process. Section 5 will present this language.

4 Test Amplification Process

In this section, we explain in detail how a given TDL test suite is amplified using our proposed approach. Figure 8 illustrates the applied process, including the proposed components and the artifacts transmitted between them. It should be noted that the overall process is indeed the same as for the conventional software (presented in Section 2.5). What differs for xDSLs is the implementation of each step as described below.

4.1 Fault Model

As it is common in the engineering field, we start by defining a fault model that makes explicit the types of mistakes that executable models may have and our proposed technique aims to uncover. To define this fault model, we take inspiration from others proposed in the literature for various domains (e. g., component-based systems [35], software code [36], graph transformation [37], adaptive systems [38]) albeit not directly applicable to xDSLs.

Our fault model considers six main types of semantic errors that may appear in executable model specifications, causing deviations from the expected behavior. These faults may have different interpretations depending on the particular xDSL, and so, next we describe them generally but also provide concrete examples for the xArduino case:

- **Missing object.** The omitted object may represent a domain concept (for instance, in the case of xArduino, it could be an undefined `DigitalPin` on a `Board`, or a missing computation instruction) or be part of the behavioral interface of the modeled system (e. g., a missing accepted event or an event with a missing parameter).
- **Extraneous object.** This type of fault corresponds to the definition of superfluous objects (e. g., unnecessary instructions or events with more parameters than necessary). While extraneous objects may not always result in incorrect runtime behavior, they are indicators of possible errors.
- **Wrong attribute value.** This may cause an object to be incorrectly characterized (e. g., the `level` of a `DigitalPin` is incorrect and therefore does not correctly reflect the state of its module) or the execution flow to be incorrect (e. g., the values in the conditional expression of a `Control` instruction are faulty, as illustrated in the `while` loop to the right of Figure 4). This category also includes incorrect attribute value assignments that can occur at runtime.
- **Missing reference.** This can lead to faulty object definitions and faulty behaviors (e. g., a `Sketch` that is not linked to the block specifying the board's behavior).
- **Extraneous reference.** Similarly to missing references, extraneous links can lead to faulty object definitions and faulty behaviors (e. g., a board with extra pins or whose behavior has unnecessary computation steps).
- **Wrong reference.** This can lead to faulty object definitions (e. g., a `PushButton` associated to an incorrect `DigitalPin`, causing the pin to inaccurately represent the button's state) and faulty execution flows (e. g., due to a wrong sequencing of computational steps).

4.2 Test Case Modification

Test cases are directed to find errors in the model under test, as defined by the fault model introduced above, by invoking events defined in the interface of the xDSL. Therefore, the first step in our test amplification process is the modification of existing test cases, which is realized by the *Test Case Modifier* component. This involves performing two tasks on each test case of the considered test suite: modifying the test input data and removing the assertions. The former aims at putting the model under test in unexplored runtime states, and the latter is required since changing the input data makes the existing assertions invalid.

For the former task, we introduce a set of test case *modifiers*. A modifier performs a single modification on the test input data of a given test case to generate a new test case. It can be applied multiple times on the same test case, yielding a different result depending on the chosen element of the test case to be modified. Each modifier may possess its own *application policy*, which tells on which possible elements

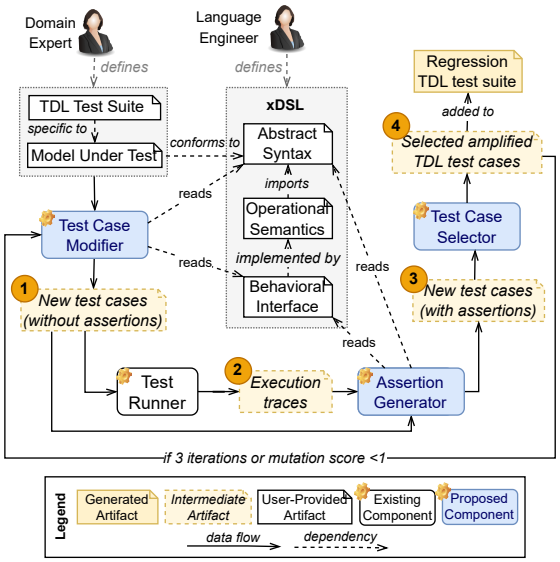


Fig. 8 Test Amplification Process

of a given test case the modifier will be applied to generate a new test case. By default, we generate a new test case for *each* possible modification. However, our approach supports further policies that will be explained in Section 5.

In the proposed approach, we use modifiers to change primitive data and event sequences, which are explained in the following two subsections.

4.2.1 Modification of Primitive Data

We adapt two existing sets of modifiers for modifying test input data that comprise primitive values: (i) the operators proposed by Danglot et al. in [18] for the amplification of JUnit test cases, and (ii) the modifiers used by the Pitest mutation testing tool [23] for putting Java programs in new runtime states. The resulting modifiers are as follows:

- A *numeric* value n is replaced by either 1, -1, 0, $-n$, $n+1$, $n-1$, $n \times 2$, $n \div 2$, or with another existing value of the same type.
- A *string* value is modified by either adding a random character, removing one of its characters randomly, arbitrarily replacing one of its characters with a random character, replacing the string with an empty string, or with a random string of the same size.
- A *boolean* value is negated.

For instance, the integer modifier will be applied 18 times on a test case containing an event occurrence with two integer parameters (2×9 possibilities).

4.2.2 Modification of Event Sequences

As explained in Section 2.3, the test input data of a TDL test case written for a model is composed of a sequence of event occurrences. According to the xDSL the tested model conforms to, such occurrences are instances of the accepted events of the xDSL's behavioral interface. Each event occurrence may have a set of parameters pointing to the model elements. Following these considerations, we propose the following modifiers to generate new test cases by modifying the input event sequences of a given test case.

- *Event Creation*: Creates new event occurrences as follows. First, this operator collects those accepted events of the xDSL's behavioral interface that (i) are not used in the given test case (indeed in its input test data), and (ii) can be instantiated (a value can be set to their parameters using information from the model under test). For each collected event, it creates all the possible event occurrences based on the values that can be set to the event's parameters. Then, it adds each new event occurrence to the input event sequences of the given test case to generate a new test case by default.
- *Event Modification*: Analyzes the model under test to find alternative values for the parameters of the events (i. e., other values of the same type). If any are found, the values of the event parameters are replaced with the alternatives.
- *Event Duplication*: Duplicates an event occurrence.
- *Event Deletion*: Removes an event occurrence.
- *Event Permutation*: Performs a random permutation of input event occurrences.

The output of this step is a set of new test cases but still without any assertion (label 1 in Figure 8).

4.3 Assertion Generation

In the second step, our *Assertion Generator* component adds assertions to the definition of the new test cases by following the idea of regression oracle checking [17]. In this technique, assertions are generated based on the execution traces of the SUT to improve the strength of the regression testing. More specifically, its objective is to improve the ability of the test

cases in detecting regressions in future versions of the SUT (not new faults). Therefore, we ensure that SUT does not violate its current and future tested behaviors.

Accordingly, we execute each new test case on the model under test using a *Test Runner* proposed in our previous work [13] to capture the execution trace of the model (label 2 in Figure 8).

Then, the *Assertion Generator* analyzes the execution trace of the model under test to generate assertions for the test case based on the model's reaction to the new test input data. As explained in Section 2.2.3, the execution trace of an executable model comprises a sequence of exposed event instances, according to the behavioral interface of the xDSL the model conforms to. Since the test cases are implemented in TDL, the assertions must be generated in TDL as well. Thus, the main role of the assertion generator is to transform the exposed event instances to TDL elements. This transformation uses the definition of both the behavioral interface and the abstract syntax of the xDSL because event instances conform to the behavioral interface and may carry EObjects of the model under test, which conform to the abstract syntax metamodel.

For example, Figure 9 illustrates a test case generated by our amplification approach from the initial test case shown in Figure 3. In the test case modification phase, the `Event Creation` modifier added a new event, `button_released (button1)`, to the input event sequence. Then, in the assertion generation phase, a new assertion with expected output `pin_level_changed (whiteLedPin == 0)` is generated because, according to Figure 2, when releasing the button, the `else` part of the Sketch will be executed, which changes the level of the `white LED` to 0. This assertion passes on the xArduino model of Figure 2 but it fails when running the generated test case against the mutated xArduino model of Figure 4.

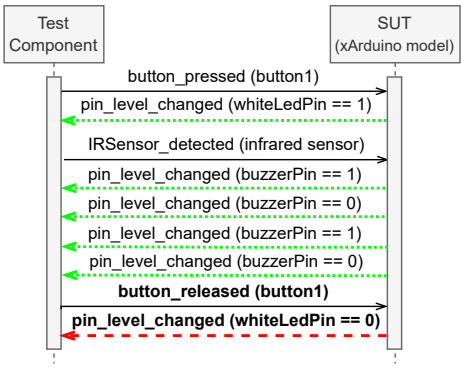


Fig. 9 Amplified TDL test case for the xArduino model of Figure 2, generated from its initial test case (Figure 3) by applying the *event creation* modifier. The last message (red dashed arrow) does not satisfy the assertion when executing against the mutated xArduino model of Figure 4.

4.4 Test Case Selection

Up to this step, a set of new test cases has been generated, but not necessarily all of them improve the quality of the input test suite. Therefore, some test quality measurement criteria must be used to filter the new test cases and keep those improving the input quality (label 4 in Figure 8). Our proposed approach supports two well-known metrics for test quality measurement: coverage and mutation analysis. Specifically, we rely on two of our previous works (already presented in Section 2.4) which compute these metrics generically for xDSLs, including:

- A *model element coverage* metric [24]; and
- The WODEL model mutation framework [33]. In this paper, we also extended WODEL to support the automatic generation of mutation operators for a given xDSL. Section 5 explains this feature.

Depending on the chosen metric, the *Test Case Selector* keeps those new test cases improving the element coverage percentage and/or the mutation score of the initial test suite. By default, the selector measures the element coverage to filter the new test cases.

For example, the new test case of Figure 9 will be added to the regression TDL test suite of the xArduino model based on both metrics because (i) it covers the else part of the Sketch of the xArduino model which was not covered by the initial test case (highlighted in red in Figure 2); and (ii) it fails when executed against the xArduino mutant of Figure 4 which means the test case is able to detect the injected fault, so the mutant is killed, hence increasing the mutation score.

Figure 10 shows in detail how a test case is selected considering mutation score improvement. First, the original model under test is mutated using a set of mutation operators for the xDSL (step 1). Then, the original test suite is evaluated on the mutants (step 2) to yield a mutation score (step 3). We also keep track of the set of mutants killed and alive. Next, the process evaluates each amplified test case on the live mutants (step 4) to check whether it fails on any of them (i. e., kills some mutant, step 5) or succeeds (i. e., does not distinguish the mutant from the original model). The amplified test case is selected if it kills at least one live mutant. In such a case, the process incorporates the newly killed mutants into the existing set of killed mutants, increases the mutation

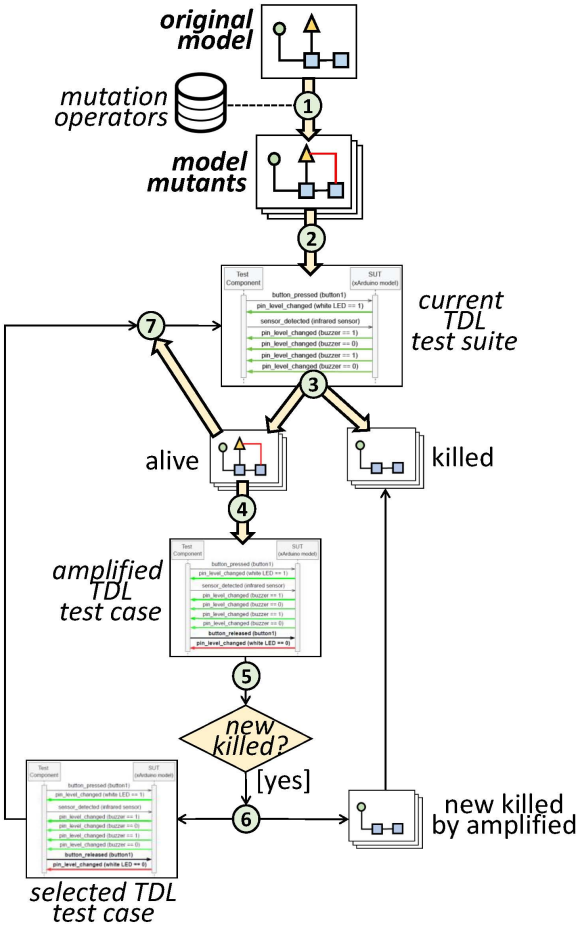


Fig. 10 Selecting test cases using mutation analysis

score (step 6), and iterates the process on the selected amplified test cases (step 7).

4.5 Stop Criteria

As can be seen in Figure 8, the test amplification process iterates on the selected new test cases based on some stop criteria (label 4). A stop criterion can be a specific number of iterations, reaching a particular test selection score (i. e., coverage percentage and/or mutation score), or a combination of both. We iterate by default up to three times while the coverage percentage is less than 100 %.

5 Test Amplification Configuration

In the proposed test amplification process, there are several variation points allowing the process customization for a given xDSL. In this section, we propose a DSL enabling language engineers to perform this custom configuration for their xDSLs.

5.1 Test AMPlification Configuration DSL (AMPC)

Figure 11 presents the concepts of our proposed DSL as a metamodel. A Configuration element refers to the metamodel and the behavioralInterface of a given xDSL and comprises three main elements.

- An AmplificationApproach that is Iterative and may specify the maximum number of iterations.
- A set of TestModificationOperators of different types, including PrimitiveDataModifier (such as NumericValueModifier, BooleanValueModifier, and StringValueModifier) and EventSequenceModifier (such as EventCreation, EventModification, EventDeletion, EventDuplication, and EventPermutation). Such operators can be applied to particular events of the behavioral interface by specifying their scope. They also have an ApplicationPolicy that determines on which possible elements of a given test case the modifier must be applied to generate a new test case, including:
  - all the possible elements at once.
  - one of the possible elements, selected randomly.
  - each of the possible elements, in which case, each modification results in a new test case.
- A Configuration element, which comprises a set of TestSelectionCriterion that can be ElementCoverage, MutationAnalysis, or a combination of both. For the MutationAnalysis, a set of *mutation operators* are needed to generate mutants from the models under test. Although these operators can be provided manually by the language engineer (UserDefinedOperator), this paper proposes new facilities for the automatic generation of mutation operators for a given xDSL (GeneratedOperators) which will be explained in the next section.

Listing 1 presents an example AMPC model using our provided textual concrete syntax. It represents our default configuration in which (i) the process is iterated up to three times (line 3); (ii) all the test

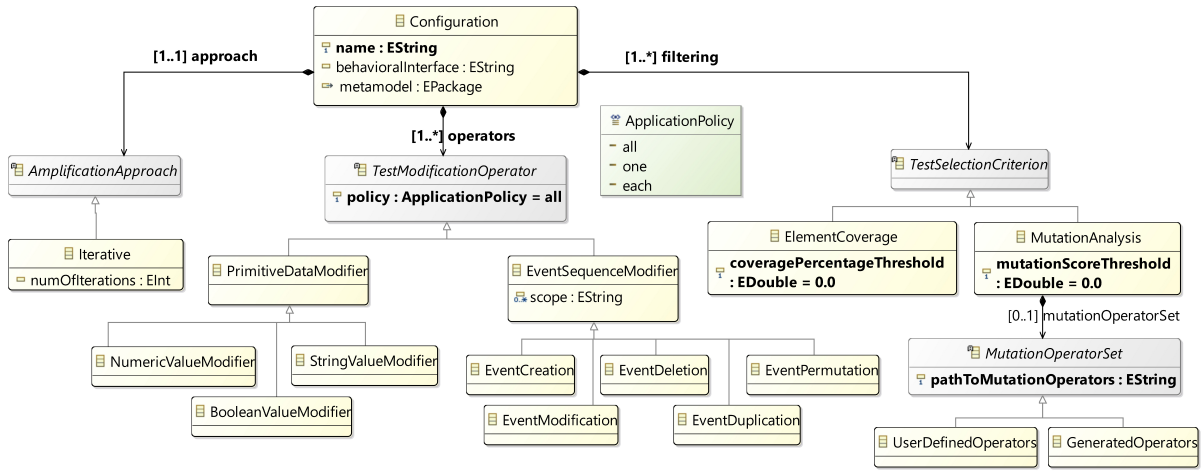


Fig. 11 Abstract syntax of the Test AMPlification Configuration DSL (AMPC)

```
1  config amplifier ArduinoAmplifier{
2      iterative technique
3          stop after 3 iterations
4
5      test modifiers{
6          all primitive modifiers on each literals,
7          all event modifiers on each events
8      }
9      filterby{ element coverage: stop at 100.0 }
10 }
```

Listing 1 Default test amplification configuration

modification operators are considered and by applying them on each possible element, a new test case will be generated (lines 5-8); and (iii) element coverage is considered for the test case selection aiming at reaching 100 % coverage (line 9).

5.2 Automatic Generation of Mutation Operators for xDSLs

As already introduced in Section 2.4.2, WODEL is a language allowing the definition of various mutation operators for a given xDSL [39]. It allows modeling the faults described in the fault model introduced in Section 4.1. In particular, it supports five different types of model mutation which can *create*, *clone*, *modify*, *retype* (i. e., change the type of an object to one of its sibling types), and *delete* elements of a model in order to generate a mutant from it. The WODEL tool ensures that the generated mutant models conform to the DSL abstract syntax and satisfy its OCL constraints.

Based on these already existing types of model mutations of WODEL, we provide a means to configure the automatic generation of mutation operators for a particular DSL. This effectively avoids the need to manually create the operators. Our approach provides an extension to the WODEL tool in the form of an automatic mutation operator generator. Figure 12 shows the metamodel defining all the concepts that can be used to configure this generator. At the top, a GeneratedOperators root element can be set in either a stochastic or an exhaustive mode, meaning that from a given model, at most a specific number of mutants (topNumOfMutants) or all the possible mutants will be generated by each operator, respectively. For a given metamodel, five different types of mutation operators (i. e., supported by the WODEL tool [33]) can be generated for different metaclasses of the metamodel. The *scope* of each requested MutationOperatorType can be defined in two ways:

- ExplicitScopeSelection: to generate operators for a set of particular metaclasses,
- ImplicitScopeSelection: to generate operators for a set of all, random, concrete, or abstract metaclasses.

Next, we explain each type of mutation operators that can be generated. Each of these types is specified as a subclass of MutationOperatorType.

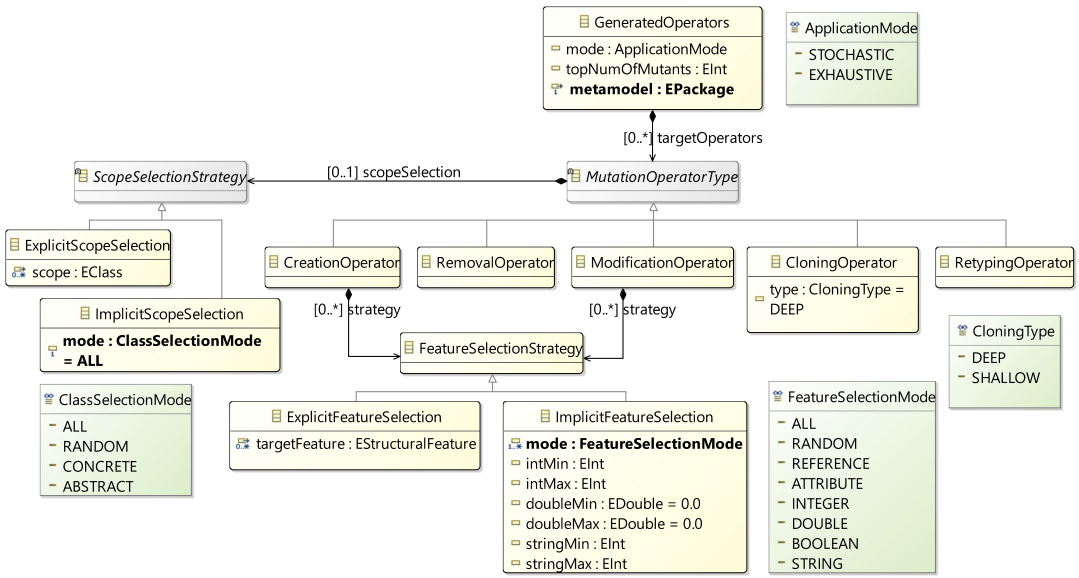


Fig. 12 Metamodel to configure the automatic mutation operator generator

CreationOperator Type

This type of mutation operator generates a mutant by creating a new object and adding it to the considered model. Such operators can be generated for *concrete* metaclasses of the given metamodel, except for the root metaclass which should never be instantiated more than once within the same model. The generated operator must also specify (i) how to initialize the mandatory features of its scope metaclass; and (ii) how to determine the container metaclass. In our default setup, we use random primitive values for the mandatory attributes and select a random valid metaclass for all the mandatory references as well as for the container. It is also possible to define a particular strategy for assigning feature values (FeatureSelectionStrategy), by considering specific features (ExplicitFeatureSelection) or implicitly all features, a random feature, all the references, all the attributes, or only integers, doubles, booleans, or strings. For example, we can specify the boundary of random values that can be assigned to numeric attributes (intMin, intMax, doubleMin, and doubleMax) or the minimum and the maximum number of characters for string attributes (stringMin, stringMax).

RemovalOperator Type

This type of mutation operator generates a mutant by removing an object from the considered model while ensuring that no dangling mandatory source/-target reference remains in the model after the object removal. This means that all the elements contained in the removed object must be removed from the model as well. Such operators can be generated for *all* the metaclasses of the given metamodel, except for the root metaclass since removing its single instance would delete the complete model. It can be noted that the WODEL mutant generator always removes objects with dangling references and no more checks are needed by the mutation operator.

Please note that even if the "*competent programmer*" hypothesis of mutation testing [25] recommends operators that perform small changes, deleting dangling edges and contained elements is necessary to avoid malformed models that would not parse otherwise.

ModificationOperator Type

This type of mutation operator generates a mutant from a model by modifying the values of the features of its objects, except their name or ID. Such operators can be generated for *all* the metaclasses of the given metamodel. Similar to the CreationOperator type, by default, we use random values for feature modification, but it is possible to define various strategies for mutating a feature value (using FeatureSelectionStrategy).

CloningOperator Type

There are two types of CloningOperators: deep and shallow. They both make a copy of an object to generate a mutant, but the former also makes a copy of all of its contained objects. Thereby, this type of mutation operator can be generated for *all* the metaclasses of the given metamodel *except* for the root metaclass.

RetypingOperator Type

This type of mutation operator changes the type of an object to one of its sibling types while preserving all the initial feature values compatible with the new target type. Consequently, this type of mutation operator can be generated for those metaclasses of the given metamodel that have a sibling metaclass.

5.3 AMPC Extension With Automatic Mutation Operator Generation

As the automatic generation of mutation operators facilitates the application of mutation analysis, its adoption in our test amplification framework will also benefit language engineers. Accordingly, we extended the AMPC DSL abstract syntax (Figure 11) with the metamodel of Figure 12 to support configuring the mutation operator generator in an AMPC model.

As an example, Listing 2 shows a custom amplification configuration model for the Arduino xDSL. We first import the metamodel (line 2) and the behavioral interface (line 3) of the Arduino xDSL to use their content later. As the test input data for the Arduino models only comprises numeric primitive values (the level feature of the DigitalPin), we define a NumericValueModifier to be applied on each possible value (line 6). Moreover, we define two EventSequenceModifiers: an EventCreation modifier to generate a new test case by adding all the missed events of a given test case, and an EventModification to generate a new test case by modifying each `button_pressed` event of a given test case (lines 7-8).

Finally, we use both element coverage (line 11) and mutation analysis (line 12) as the test selection criteria aiming at reaching 100 % coverage and 85 % mutation score. For the mutation analysis, we specify what kind of mutation operators to be generated for which metaclasses of the Arduino xDSL, including creation operators for the concrete classes, deep cloning operator for the ModuleAssignment class, and modification operators for all the classes by mutating their integer attributes with random values between 0 and 10 (lines 13-19). The generated mutation operators must be applied exhaustively on the model under test to generate all the possible mutants (line 13).

```
1  config amplifier ArduinoAmplifier{
2    import metamodel arduino
3    import behavioral interface "Arduino.bi"
4    iterative technique
5    test modifiers{
6     modify each numeric values,
7     create all missed events,
8     modify each ["button_pressed"] events
9    }
10   filterby{
11     element coverage: stop at 100.00,
12     mutation score: stop at 85.00 {
13       generate exhaustive mutants using mutation operators
         "/ArduinoOperators.mutator" {
14           creation operators for [concrete classes],
15           deep cloning operators for [ModuleAssignment],
16           modification operators for [all classes]{
17             modify integers with values between 0 and 10
18           }
19         }
20       }
21     }
22 }
```

Listing 2 Customized test amplification configuration for Arduino xDSL

Listing 3 shows an excerpt of a WODEL program generated automatically for the xArduino metamodel based on the configurations specified in lines 14-18 of the AMPC model (Listing 2). The ApplicationMode of the generated mutation operators is exhaustive (line 1 of Listing 3) as requested by the AMPC model (line 13 of Listing 2). Here we show three of the generated mutation operators: (1) a creation mutation operator for the BinaryBooleanExpression which sets its attributes and references to random values (lines 5–10); (2) a deep cloning mutation operator for the ModuleAssignment (lines 11–13); and (3) a modification mutation operator for the IntegerConstant which modifies its value feature with a random integer between 0 and 10 (as indicated in line 18 of Listing 2).

6 Tool Support

We implemented our approach in the form of a tool integrated into the Eclipse GEMOC Studio [27], a language and modeling workbench for xDSLs defined on

```
generate exhaustive mutants in "data/out/"
from "data/model/" metamodel "http://xArduino/1.0"

with blocks {
b0 {
  create BinaryBooleanExpression
    with {operator in ['sup', 'inf', 'infOrEqual',
      'supOrEqual', 'equal', 'AND', 'OR', 'Different']},
      left = one Expression, right = one Expression}
}
b1 {
  deep clone one ModuleAssignment
}
b2 {
  p = modify one IntegerConstant
    with {value = random-int(0, 10)}
}}
```

Listing 3 Excerpt of the generated WODEL program.

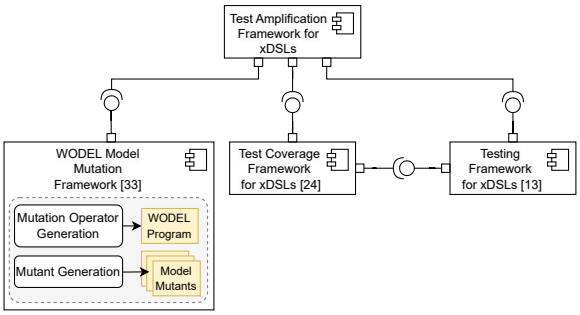


Fig. 13 Tool Architecture

top of the Eclipse Modeling Framework (EMF) [30]. The AMPC DSL is implemented as follows: the abstract syntax is defined using an Ecore metamodel, the textual concrete syntax is implemented as an Xtext grammar [40] along with scope providers, and the execution semantics is implemented in Java. Figure 13 shows the high-level architecture of the developed tool. We conveniently reused the tools implemented for our previous approaches, including our testing framework [13], our coverage computation approach [24], and the WODEL model mutation framework [33] as they are all implemented based on EMF technologies.

The WODEL interface is extended in this paper to provide two functionalities for external tools: the mutation operator generation and the mutant generation. In this paper, they are both used by our proposed test amplification framework as follows. When an AMPC model requests the automatic generation of mutation operators for a given xDSL, it is forwarded to the *Mutation Operator Generation* which specifies which kinds of mutation operators must be generated for which metaclasses of the given metamodel and with which strategy (if any). The result is a WODEL program containing the generated mutation operators which will be saved on the disk for later use. The extended functionality of the WODEL tool also allows the programmatic execution of a given WODEL program for the generation of the mutants from the conforming models.

The domain experts can use our provided test amplification tool to request the amplification of an existing TDL test suite for a given model. The tool has a default configuration but it can also be customized by the language engineers through defining an AMPC model for an xDSL. Once the tool execution finishes, the result will be serialized in two main files: one containing the new test cases generated by the amplification process (if any) and another reporting the results of the test selection phase. Let us mention that the content of the reports depends on the metric selected in the configuration: *coverage* and *mutation analysis* metrics. The former report includes the initial and final coverage percentages, the elements covered by the input and output test cases, and the elements that are still not-covered despite the amplification. The latter contains the initial and final mutation score, the mutants killed by each test case, and the mutants that are still alive despite the amplification.

The source code is available on a Zenodo repository [41].

7 Evaluation

In this section, we evaluate our proposed approach, aiming to answer the following research questions (RQs):

RQ1 To what extent the framework is applicable to xDSLs of various application domains?

RQ2 To what extent do the generated test cases increase the initial score of the original, manually-written, test cases?

RQ3 To what extent do the *size* and the *quality* of the original test suites impact the amplification result?

Table 1 Evaluation Setup

		xArduino	xPSSM
xDSLs	Abstract syntax size (#EClasses)	59	39
	Operational semantics size (LoC)	768	975
	Behavioral interface size (#Events)	7	4
xModels & Tests	Number of tested models	6	65
	Size range of xModels (#EObjects)	18-59	13-154
	Initial test suite size (#test cases)	22	216
Mutation Analysis	#mutation operators	36	29
	#generated mutants	394	12,087

RQ4 How efficient are the generated mutation operators?

In the following, we present the experiment setup and the answer to each RQ (Section 7.1), and describe threats to the validity of the experiments (Section 7.2). The evaluation data is also accessible from the same Zenodo repository [41].

7.1 Experiment Setup & Result

Setup for RQ1

For the first research question, we intend to evaluate whether the proposed framework can provide test amplification facilities to dissimilar xDSLs. In our experiment, we consider two xDSLs from two different application domains. The first xDSL is xArduino, already presented in Section 2.2. The second is xPSSM, an executable DSL for simulating systems with discrete-event behavior using the Precise Semantics of UML State Machines (PSSM) standard of the Object Management Group (OMG) [1]. Table 1 summarizes the size of these two xDSLs by reporting on the number of classes of their abstract syntax, the number of Lines of Code (LoC) of their operational semantics, and the number of events of their behavioral interface.

For each xDSL, we prepared a set of models of different sizes, summarized in Table 1. Specifically, we manually defined 6 xArduino models with sizes ranging from 18 to 59 objects and 5 xPSSM models with sizes between 13 and 154 objects. For xPSSM, the PSSM standard provides a set of UML state machine models, each one with a small test suite (one test case per model) for asserting that a given PSSM implementation executes the models in compliance with the standard [1]. Within this set, we identified a subset of models and test cases that focus on the event-driven behavior of state machines, as expected in the xDSLs considered in our approach. This subset comprises 60 models with sizes between 13 and 69 objects. Hence, overall we considered a total of 65 xPSSM models (5+60).

As for the manually-written test cases for the considered models, we provided one test suite per model, cumulatively 71 test suites:

- 6 test suites for the xArduino models, altogether containing 22 test cases,
- 65 test suites for the xPSSM models, altogether containing 216 test cases where 60 of them are the TDL versions of the test suites provided by the PSSM standard [1].

To evaluate the mutation analysis support of the approach, we manually defined a set of mutation operators for each xDSL using the WODEL framework [33]. We created 36 operators for xArduino, which only mutate the behavioral part of the models and ignore the physical-related concepts. For xPSSM, we defined 29 mutation operators, based on previous work on state machine mutation [33, 42–45]. Overall, the metamodel footprint [46] of the mutation operators covers 69.56 % of the xArduino metamodel and 50 % of the xPSSM metamodel. We used the WODEL mutant generator [39] to apply the defined mutation operators on the considered models. WODEL generated 394 and 12,087 mutants for the xArduino and the xPSSM models, respectively.

Answering RQ1

The purpose of the first research question is to assess whether our proposed approach can be used by various xDSLs to amplify the test cases of their conforming models. To answer this question, we used the prototype presented in Section 6 for the two considered xDSLs. We configured the test amplification process in two modes by defining two AMPC models. They both applied the default test case generation approach (as presented in Section 5.1) but each with a different test case selector: one considering coverage improvement up to 100 % and another aiming for reaching 100 % mutation score. We then executed the prototype on the 71 test suites on both configurations.

- **Amplification result for element coverage improvement**: Among the 71 test suites, 39 of them needed improvement as their initial coverage percentage was less than 100 %. For 20 of them, the approach was able to improve the test suite coverage by generating 32 new test cases.
- **Amplification result for mutation score improvement**: All the 71 test suites needed improvement as their initial mutation score was

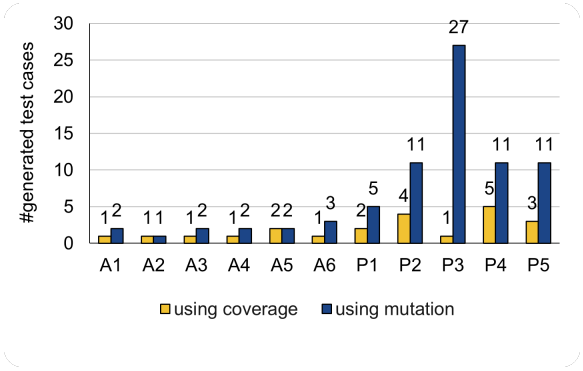


Fig. 14 Overall test amplification result for the manually defined xArduino and xPSSM models (the A and P bars, respectively).

less than 100 %. For 67 of them, the approach was able to improve the initial mutation score by generating 209 new test cases.

For example, Figure 14 shows the number of generated test cases for the manually defined xArduino and xPSSM models (the A and P bars, respectively). It is evident that for most of the cases, the number of generated test cases is different when targeting different goals i. e., coverage or mutation score improvement. We performed more analyses to identify why the obtained results are different while both AMPC models applied the same test case generation process. In our analysis, we observed that the test cases generated for improving mutation score include those generated for coverage improvement as well as other test cases which were able to kill some live mutants, hence increasing the mutation score. However, as these additional test cases did not increase the coverage percentage, they were indeed filtered by the AMPC model considering coverage improvement. For example, we investigated the P3 xPSSM model, since there is a big discrepancy in its generated test cases between coverage (1 test case) and mutation (27 test cases). We observed that it is the only xPSSM model with boolean-guarded transitions and many of its live mutants were those having faults on such transitions. The majority of its generated test cases for mutation score improvement were indeed produced by applying the *boolean test data modifier*, hence being able to kill the mentioned live mutants with no more progress on coverage improvement.

Finally, we can conclude that our proposed approach does provide a certain level of generality, i. e., the approach is not solely dedicated to a single specific xDSL, and can be applied to at least two different ones.

Setup for RQ2

In our answer to RQ1, we presented the effectiveness of the proposed framework in improving the test suites of the models of two different xDSLs. In addition, the degree of achieved improvement by the generated test cases must be analyzed to measure the framework efficiency, which is the target of RQ2, starting with the same setup as RQ1.

Answering RQ2

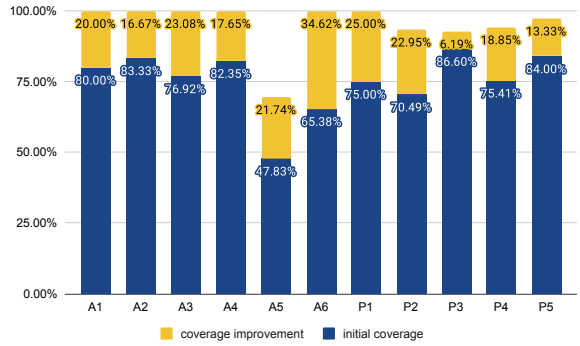
To answer the second research question, we must evaluate the degree of improvement gained by the generated test cases. Figure 15 presents the results for the provided 71 test suites using bar charts. The **(a)** and **(b)** charts display the result for the 11 test suites of manually defined models—6 xArduino models (A bars) and 5 xPSSM models (P bars)— targeting coverage and mutation score improvement, respectively. We have obtained improvements for all the test suites:

- **(a)** coverage percentage improvement between 6.19 % (P3) and 34.62 % (A6) and reaching to 100 % for 6 test suites (A1-A4, A6, and P1).
- **(b)** mutation score improvement ranging from 4.06 % (P4) to 55.10 % (A4) and reaching to 100 % for 5 test suites (A1-A4, and A6).

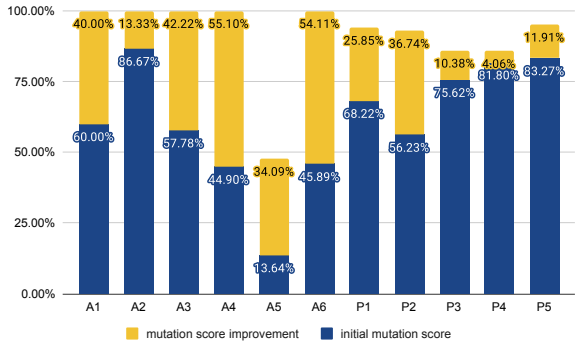
Among the xArduino test suites, the achieved improvement for A5 is considerably lower than the others (A1–A4 and A6, all reaching 100 %). By investigating the A5 model, we observed that it has an `If` block that can be entered once the level of a `DigitalPin` element is greater than 128, which is a very specific value. On the other hand, its manually written test suite did not have any test case with test input data related to this part of the model. Consequently, the test amplifier would not be able to produce any test case which can reach the body of the mentioned `If` block. In RQ3, we will discuss in more detail how the input test suite can impact the amplification result.

The **(c)** and **(d)** charts of Figure 15 show the result for the 60 test suites from the PSSM standard [1]; where each test suite has only one test case.

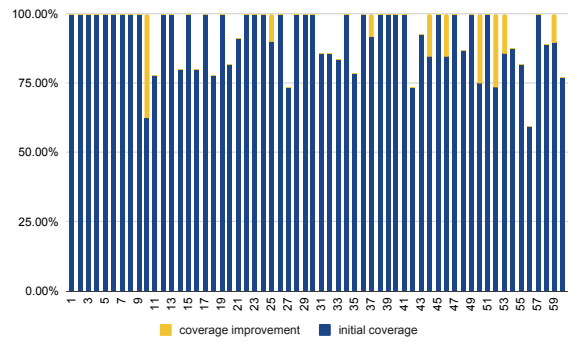
- **(c)** among the 28 cases with initial coverage percentage less than 100 %, the coverage of 9 test suites is improved, each reaching to 100 %.



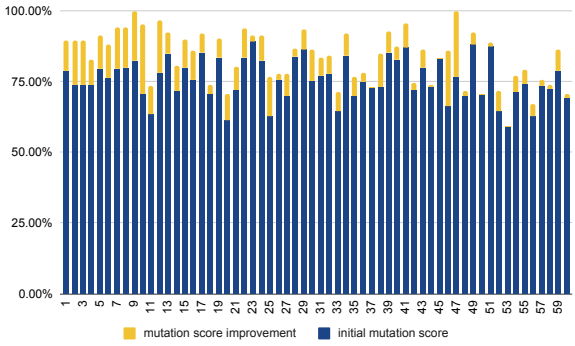
(a) Coverage percentage improvement for manually-provided data set



(b) Mutation score improvement for manually-provided data set



(c) Coverage percentage improvement for existing data set



(d) Mutation score improvement for existing data set

Fig. 15 Score improvement by test amplification

- **(d)** except for 4 cases (bars 37, 45, 50, and 53), the mutation score is improved, reaching 100 % for 2 test suites (bars 9 and 47).

These results demonstrate that, even when starting from small test suites with just one test case, the approach can provide improvement. Therefore, it reveals the success of our test amplification approach in improving the models' test suites. However, the rate of improvement is different from case to case, as we will discuss while answering RQ3.

Setup for RQ3

The third research question aims to investigate whether the *size* and the *quality* of the initial test suite impact the amplification result. In this regard, we classified our provided TDL test suites into 4 categories considering their mutation score as the test quality metric:

1. Small Size Medium Quality (SSMQ): having one test case with mutation score $< 80\,\%$
2. Small Size High Quality (SSHQ): having one test case with mutation score $\geq 80\,\%$
3. Medium Size Medium Quality (MSMQ): having more than one test case with mutation score $< 80\,\%$
4. Medium Size High Quality (MSHQ): having more than one test case with mutation score $\geq 80\,\%$

We further populated our experiment data for this RQ by applying the following automated process on each MSHQ test suite: we randomly removed some of its test cases to achieve a new test suite with a mutation score < 80 %. Then, we classified this new test suite in (i) SSMQ if it had only one test case; or (ii) MSMQ if it had more than one test case. The rationale for selecting 80 % as the threshold to classify a test suite as having medium or high quality is because improving the mutation score beyond 80 % is time-consuming for developers [47].

Table 2 Evaluation result for amplifying test suites with different sizes and qualities based on the mutation score

		# Orig. test suites	Orig. test suites size (Total # test cases)	# Generated mutants	# Killed mutants by orig. test suites	Orig. mutation score	# New ampl. test cases	# New killed mutants by ampl. tests	Mutation score improvement	Regression test suite size (# orig. + # ampl.)	Regression test suite global score (orig. + improv.)
SSMQ	**xArduino**	4	4	148	60	44.08%	8	65	42.85%	12	86.93%
	xPSSM	43	43	5,653	4,022	71.99%	101	400	8.69%	144	80.68%
SSHQ	**xArduino**	1	1	15	13	86.67%	1	2	13.33%	2	100.00%
	xPSSM	18	18	2,205	1,873	84.38%	36	133	7.13%	54	91.51%
MSMQ	**xArduino**	1	2	231	106	45.89%	3	125	54.11%	5	100.00%
	xPSSM	4	44	4,229	3,385	74.23%	55	459	14.26%	99	88.49%
MSHQ	**xArduino**	2	6	241	222	91.10%	3	19	8.90%	9	100.00%
	xPSSM	5	156	4,453	4,249	93.28%	47	88	3.17%	203	96.45%

For every category and xDSL, Table 2 presents the number of original test suites (column 3), the size of original test suites as the cumulative number of their test cases (column 4), the number of generated mutants (column 5), the number of mutants killed by the original test cases (column 6), and the original mutation score (column 7). The provided numbers are cumulative, and the scores are the average scores considering all provided models and test suites.

Answering RQ3

Given a test suite to be amplified, its *size* (i. e., number of test cases) and *quality* (i. e., coverage percentage or mutation score) may influence the level of improvement that our approach provides. The third research question targets this matter and to answer it, we run the experiment on the test suites of the four considered categories. We configured the test amplification process (Figure 8) (i) to apply all the possible modifications on every given test case to generate all candidate new test cases; and (ii) to continue up to 3 iterations while the mutation score is less than 100%. Therefore, the only constraint on the execution time is the number of iterations to avoid a huge experimentation time.

Table 2 lists detailed numbers related to our experiment. First, we compare the results for the test suites of the same size but with different qualities (SSMQ vs SSHQ, and MSMQ vs MSHQ). As can be seen, the number of new test cases (column 8) and the average improvement (column 10) for high-quality tests is less than the one for medium-quality tests. This is due to the fact that high-quality test suites need less improvement. However, the final mutation score of the regression test suite (last column) – the sum of the original score and the score improvement – is higher for high-quality tests. For example, for xArduino, the mutation score of SSMQ test suites is improved from 44.08 % to 86.93 % (42.85 % improvement), but for the SSHQ ones, it is improved from 86.67 % to 100 % (13.33 % improvement). Note that the scores refer to the average score of all considered test suites in each category. Also for xPSSM, the mutation score for the SSMQ test suites augments from 71.99 % to 80.68 % (8.69 % improvement), and for the SSHQ ones from 84.38 % to 91.51 % (7.13 % improvement). By investigating the result, we observed that an input test case with a higher initial score usually has more diverse test input data. For example, its sequence of input event occurrences is larger, and each event occurrence conforms to a various event with different event parameter values. Consequently, more modification operators can be applied to it, resulting in generating a higher number of test cases and having a better chance of finding more effective ones. We compare the effectiveness of two new test cases based on their impact on quality improvement, such as measuring how much each test case augments the mutation score of the input test suite.

Second, we compare the results for the test suites with different sizes but similar qualities (SSMQ vs MSMQ, and SSHQ vs MSHQ). According to the numbers in the last column, the final mutation score is higher when the original test suite has more test cases. For instance, comparing the final mutation score of the xArduino test suites, for SSMQ is 86.93 % while for MSMQ is 100 %, and for both SSHQ and MSHQ

is 100 %. Likewise, for the xPSSM test suites, the final mutation score is 80.68 % for SSMQ but 88.49 % for MSMQ, and 91.51 % for SSHQ but 96.45 % for MSHQ. Therefore, for test suites with more test cases, it appears that there is more chance to generate new test cases improving the mutation score of the test suite. A possible explanation is that our approach runs the amplification on *every* test case of the original test suite, each time by applying all the possible modifiers to generate as many new test cases as possible until no more progress is possible. Hence, the more available initial test cases, the more generated test cases, and the more chances for improving the test suite quality.

However, a question may arise regarding the execution time, whether more execution time would impact the result. Based on the applied configuration, a longer execution time can only be seen as a higher number of iterations (>3) if possible (i. e., if an input is available for the next iteration). To answer this question, we carefully checked the results of each category to identify if there was any case stopping at the third iteration without reaching 100 % mutation score – needing more iterations – and we found none. This means assigning more execution time would not change the experiment result.

Setup for RQ4

With the last research question, we aim at investigating the efficiency of our proposed mutation operator generator from two perspectives: the language engineer's point of view and the metamodel coverage. For the former, we used the mutation operators manually defined in the setup for RQ1 to be compared with the automatically generated ones from default and customized AMPC configuration models. For the latter, the WODEL tool helps in evaluating the coverage of the operators w. r .t the metamodel [48].

Answering RQ4

To answer RQ4, we have compared the mutation operators created by hand in the evaluation setup of RQ1 with those generated automatically for both xArduino and xPSSM in two different experiments:

- *Customized automatic*. We executed the automatic mutation operator generator with a custom AMPC configuration, in which we aimed at evaluating whether the generator can provide operators emulating the manually defined ones.
- *Fully automatic*. To assess the ability of the generator in providing effective mutation operators, we executed it in its default configuration in which all five different types of mutation operators (creation, cloning, modification, retyping, and removal) are expected to be generated for all the concrete metaclasses and considering all of their features.

Table 3 presents the results of our experiments, including the percentage of the metaclasses affected by the mutation operators to perform *creation*, *modification*, and *deletion* operations on the models to generate mutants.

Table 3 Comparison of the mutation operators created by hand and those generated automatically

	xArduino			xPSSM		
Metamodel classes coverage	**By hand**	**Customized automatic**	**Fully automatic**	**By hand**	**Customized automatic**	**Fully automatic**
Creation	0%	0%	100%	16%	16%	96%
Modification	43%	43%	97%	20%	20%	96%
Deletion	0%	0%	100%	10%	10%	96%
#mutation operators	36	46	226	29	46	187
#generated mutants	394	362	1,266	12,087	18,971	35,978

For both xArduino and xPSSM, the *customized automatic* approach was able to generate mutation operators with the same metamodel coverage as the manually defined ones, but with a higher number of operators: 46 operators in each case, in comparison to 36 and 29 operators defined *by hand* for the xArduino and xPSSM, respectively. We also applied the generated operators on the 71 considered models and we achieved a similar number of mutants with slight differences.

Moreover, the *fully automatic* approach successfully generated a set of mutation operators for both xArduino and xPSSM (226 and 187 operators, respectively) with almost 100% metamodel coverage (root and abstract metaclasses are excluded from some operators). The operators were able to generate 1,266 xArduino mutants and 35,978 xPSSM mutants, which is almost three times more than the result produced by the *by hand* and *customized automatic* operators. These results demonstrate the efficiency of our proposed approach for the automatic generation of mutation operators.

7.2 Threats to Validity

In this subsection, we identify threats to the validity of our evaluation according to the four main categories defined by Wohlin et al. [49]. We also explain how we mitigated each threat.

External Validity

Although we tried to apply the approach on two xDSLs from completely different domains, there is still an external threat to the generality of our approach. Therefore, we would like to apply our approach to more xDSLs in the future. Following the same direction, as the granularity of the behavioral interface of the xDSLs has a direct impact on the diversity of the test cases of their conforming models, we plan to explore the approach on xDSLs with more complex behavioral interfaces.

Internal Validity

We iterated the amplification process while there is progress, meaning that the input for the next iteration is the output of the current iteration, i. e., only those new test cases improving the score (label 4 in Figure 8). Without this test case selection criterion (i. e., iterating from label 3 of Figure 7), the number of generated test cases increases exponentially after a few iterations. However, the non-effective test cases (i. e., test cases not contributing to improving the mutation score in the current iteration) might become effective in the next iterations since a combination of several modifiers is applied to them.

For test case modification, we used the modifiers presented in Section 4.2. Other more complex modifiers could be devised, which may show more effectiveness. We plan to investigate this in future work, but we have shown that our proposed modifier set is enough to find effective amplified test cases.

Construct Validity

In our answer to RQ4, we demonstrated promising results regarding the relativeness and the metamodel coverage of the generated mutation operators. As these operators are used for the generation of mutants which are then applied for the test quality measurement, they directly impact the mutation analysis result. Therefore, a more thorough study must be conducted to compare the mutants generated by each set of operators (i. e., manually defined vs generated). For example, one can measure the mutation score of the test suites, once using the mutants generated by the manually defined operators and another time using those produced by the generated operators, and compare the results.

Conclusion Validity

Usually, amplified test cases must be approved by the developers who wrote the original test cases. Accordingly, there is a need for a user-centric evaluation to assess the value of the generated TDL test cases from the user's perspective.

In addition, we have shown the success of our proposed approach in amplifying the models' test cases. However, a performance evaluation is needed to identify the cost of using the tool, especially in its different configurations, such as comparing the execution time when targeting coverage improvement against targeting mutation score growth.

8 Related Work

This section provides an overview of related research regarding test input data modification (Section 8.1), test amplification for regression testing (Section 8.2), test case generation for behavioral models (Section 8.3), and mutation operators for DSLs (Section 8.4), identifying the innovations of our approach.

8.1 Test Input Data Modification

Data mutation testing [50] is a method inspired by the classical mutation testing for generating large test suites from a seed of a small set of test cases. The difference lies in how and where the mutation operators are applied. In mutation testing, the mutation operators are applied to the source code of a program to measure the adequacy of the test suite. Instead, data mutation applies mutation operators to the input data for generating test cases.

In the last years, this method has been applied for different purposes [51–53]. Sun et al. [51] propose a methodology for generating metamorphic relations. These relations are created by applying data mutation in the input relation. Then, a combination of constraint validation and generic mapping rules is used to generate output relations. Similarly, Zhu [53] introduces JFuzz, an automated framework for Java unit testing that combines data mutation and metamorphic testing for deriving and expressing metamorphic relations. Xuan et al. [52] present a proposal for detecting

program failures by reproducing crashes through data mutation. In contrast to the previous approaches, their work does not focus on generating new test cases, but on updating the existing ones for triggering crashes on the program under study and therefore, finding errors.

Generating input test cases is also essential for fuzzy testing [54], which consists of generating random input data as a test case, and monitoring the program for crashes or failing assertions. Fuzzers–the programs generating the inputs–can generate new inputs from scratch or modify existing ones using data mutation.

Innovation of our approach. Compared to these approaches, our input modifiers consider both primitive data and event sequences, where in addition, event parameters are model objects.

8.2 Test Amplification for Regression Testing

Several approaches use test amplification for regression testing. Xie [17] presents a framework for augmenting test suites with regression oracle checking. This proposal is supported by a tool, called Orstra, which focuses on asserting the behavior of JUnit test cases. For this purpose, Orstra amplifies automatically generated test cases by systematically adding assertions for improving their capability of avoiding regressions. DSpot [18] targets the automatic amplification of JUnit test cases. It combines input space exploration [55] with regression oracle generation [17] techniques. The former is applied for putting the program under test in never explored states, and the latter aims at generating assertions for those new states. Given a set of manually-written JUnit test cases, DSpot generates variants of them that improve the mutation score. On the basis of DSpot, Abdi et al. [19] propose Small-Amp, an amplification approach for the Pharo Smalltalk ecosystem. Ebert et al. [18, 20] provide a test amplification tool for Python. To this aim, the authors rely on the DSpot design, combining with Small-Amp features to alleviate the shortcomings related to dynamically typed languages.

Assis et al. [56] present an approach for test amplification of cross-platform applications. For this, the authors use four test patterns that analyze well-known features of a mobile application. Similarly to our approach, the test input data is a sequence of events that is exchanged with the SUT, but their input modifiers are defined using a set of test patterns specific to mobile applications.

Innovation of our approach. In general, all existing amplification tools target programs implemented by general-purpose languages such as Java [18], Pharo Smalltalk [19], and Python [20]. In contrast, our proposal supports executable models defined by xDSLs.

8.3 Test Case Generation for Behavioral Models

Some researchers have used model-driven engineering (MDE) or other automated means to generate test cases from modeling artifacts, most notably from requirement models, use cases, or activity diagrams [57–60]. Most of these efforts follow one of two main approaches for test case generation: path/coverage analysis [58, 59], or category partition [59, 60]. The former approach is based on analyzing all possible paths of behavior in the source model, and the latter partitions the requirements under test and generates test cases for combinations of such partitions. Differently from us, these efforts are specific for models of system functional requirements, they do not assume an initial set of test cases, and do not propose any test case improvement technique.

Outside requirements modeling, test case generation for behavioral models has been handled using different methods. For example, Frolich and Link [61] generate test cases from Statecharts by translating the Statecharts into a planning problem and using a planning tool to find test cases as solutions to the problem. Ahmadi and Hili [62] present an approach to automatically test components of UML-RT models with respect to a set of properties defined by state machines, and apply slicing to reduce the size of the components with respect to the properties. Rocha et al. [63] generate JUnit test cases from sequence diagrams via a transformation of the latter into extended finite state machines. From fUML activity diagrams, Iqbal et al. [9] generate test cases with input data to cover all executable paths of the diagrams, together with their expected output. The interested reader can consult [64] for a recent survey on model-based testing using activity diagrams, including test case generation. In summary, test case generation for behavioral models has been tackled in the literature, but the proposals are normally language-specific, while our approach aims to be generic. Moreover, these proposals do not target test amplification (i. e., improving an existing test suite).

Also in the modeling area, some research efforts focus on test case generation for model transformations, or transformation models. A test case in this scenario comprises an input model to the transformation and an oracle function. For example, Giron et al. [65] use software product lines and input metamodel coverage to generate a reduced set of test cases for transformations; Guerra and Soeken [66] use constraint solving to generate test models and partial oracles from declarative transformation specifications; Al-Azzoni and Iqbal [67] apply test case prioritization for regression of transformations based on an analysis of the transformation rules' coverage; and Troya et al. [68] infer likely metamorphic relations for ATL transformations, which can be used for metamorphic testing. In a similar way as we analyze the test execution traces to derive test assertions, Troya et al. rely on the traces of the transformation executions to derive the metamorphic relations. However, their goal (metamorphic testing) and ours (amplification) are different.

Innovation of our approach. Altogether, we find a variety of approaches for test case generation in the modeling area, but to our knowledge, ours is the first one targeting *test suite* improvement for xDSLs. Our test amplification proposal can be used as a complement to these existing test case generation approaches to improve the quality of their generated test suites for regression testing.

8.4 Mutation operators for DSLs

Our approach supports mutation analysis to evaluate the quality of test cases. For this purpose, we enriched WODEL with a DSL to configure the automatic generation of mutation operators for a target DSL. Other approaches to define and generate mutation operators for DSLs exist, which we analyse next.

A trend of works are related to search-based software engineering, where optimisation problems are represented as models (conforming to a metamodel), and search algorithms are used to find solutions, requiring mutation and crossover operators, represented as rules. This approach is supported by tools such as MOMoT [69], Fitness Studio [70], MDEOptimiser [71], or ViatraDSE [72].

A common challenge in this area is to automate the generation of mutation operators that yield good results for the problem being solved. This way, Burdusel et al. [71] propose a mechanism to generate multiplicity-preserving search mutation operators for a given optimisation problem. The operators have the form of Henshin graph transformation rules [73]. In a similar vein, van Harten et al. [70] propose a framework for generating mutation operators for multi-objective problems. Similarly, operators are expressed as Henshin rules, but the search can combine generated and manually-created operators.

Instead, WODEL is a textual language explicitly tailored for mutation, and we now provide a DSL to configure the generation of operators. Our focus is on testing, not on search-based problems. Similar to [70], evaluating amplified tests may use automatically generated operators, or existing ones.

Model-based testing is also considered in related work on mutation analysis. For example, we synthesize mutation operators for MOMoT [74], which are used to generate follow-up test cases in a metamorphic testing environment. Some approaches – like our own WODEL-TEST [33] – use model-driven techniques for mutation analysis. Semeráth et al. [75, 76] propose techniques to generate diverse models, which then can be used as input cases for model-based testing. The authors reported good mutation scores using such technique to generate state machine and wind turbine control system models, which are used as test cases to detect injected faults in well-formedness constraints within the metamodel, and in graph pattern rules. However, the mutation operators are fixed. Alhwikem et al. [77] define a set of abstract mutation operators, which can then be used to define concrete operators for languages defined via a metamodel. However, the expressivity of the operators is limited (e. g., changing features, or creating objects).

Innovation of our approach. As part of the amplification process support, we have created a novel DSL to customise the generation of model-based mutation operators, to be used in mutation testing, which is a novel contribution of this work.

9 Conclusion and Future Work

In this paper, we have presented a test amplification framework for executable DSLs. The framework offers (i) a generic test amplification process for amplifying TDL test cases of executable models; (ii) the AMPC DSL, a language for custom configuration of the proposed process for a given xDSL; and (iii) an automatic approach for the generation of mutation operators for a given xDSL which facilitates the usage of mutation score as the test selection criterion. We

propose tool support atop the Eclipse GEMOC studio and report on an evaluation on two different xDSLs that shows the success of the framework in improving the effectiveness of the test cases, as given by both the coverage percentage and the mutation score.

As possible interesting future work, we believe the whole amplification process could be recast as a search process, e. g., based on genetic algorithms. This would require additional work: crossover operators for the TDL test cases would need to be defined, input modifiers would be used as mutation operators, and the fitness function would be driven by coverage percentage or mutation score improvement. With regards to the evaluation, more experiments can be conducted to answer other interesting research questions, such as which are the most effective test modifiers or how much the framework meets the user's expectations regarding usability and cost. Finally, given the promising results obtained so far, we aim at investigating test case amplification for other purposes than regression testing.

Acknowledgments. This project has received funding from the European Union's Horizon 2020 research and innovation program under the Marie Skłodowska Curie grant agreement No 813884, the Spanish Ministry of Science (PID2021-122270OB-I00 and TED2021-129381B-C21), and the joint program of Young Doctors from the Madrid region and the Universidad Autónoma de Madrid (SI4/PJI/2024-00191).

References

[1] Object Management Group (OMG): Precise Semantics of UML State Machines. https://www.omg.org/spec/PSSM/1.0/About-PSSM/. (last accessed in February 2024) (2019)

[2] (OMG), O.M.G.: Semantics of a Foundational Subset for Executable UML Models. https://www.omg.org/spec/FUML/. (last accessed in February 2024) (2021)

[3] Bendraou, R., Combemale, B., Crégut, X., Gervais, M.-P.: Definition of an eXecutable SPEM 2.0. In: 14th Asia-Pacific Software Engineering Conference (APSEC), pp. 390–397. IEEE Computer Society, Nagoya, Japan (2007). https://doi.org/10.1109/ASPEC.2007.60

[4] OASIS: Web Services Business Process Execution Language Version 2.0. https://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html. (last accessed in February 2024) (2007)

[5] Erdweg, S., Storm, T.v.d., Voelter, M., Tratt, L., Bosman, R., Cook, W.R., Gerritsen, A., Hulshout, A., Kelly, S., Loh, A., Konat, G., Molina, P.J., Palatnik, M., Pohjonen, R., Schindler, E., Schindler, K., Solmi, R., Vergu, V., Visser, E., Vlist, K.v.d., Wachsmuth, G., Woning, J.v.d.: Evaluating And Comparing Language Workbenches: Existing Results And Benchmarks For The Future. Computer Languages, Systems and Structures **44**(Part A), 24–47 (2015) https://doi.org/10.1016/j.cl.2015.08.007

[6] Mijatov, S., Mayerhofer, T., Langer, P., Kappel, G.: Testing functional requirements in uml activity diagrams. In: Blanchette, J.C., Kosmatov, N. (eds.) Tests and Proofs, pp. 173–190. Springer, Cham (2015)

[7] Kos, T., Mernik, M., Kosar, T.: Test automation of a measurement system using a domain-specific modelling language. Journal of Systems and Software **111**, 74–88 (2016)

[8] Lübke, D., Lessen, T.: BPMN-based model-driven testing of service-based processes. In: Enterprise, Business-Process and Information Systems Modeling, pp. 119–133. Springer, Berlin, Germany (2017)

[9] Iqbal, J., Ashraf, A., Truscan, D., Porres, I.: Exhaustive simulation and test generation using fuml activity diagrams. In: Proceedings of the 31st International Conference on Advanced Information Systems Engineering (CAiSE). Lecture Notes in Computer Science, vol. 11483, pp. 96–110. Springer, Rome, Italy (2019)

[10] Wu, H., Gray, J., Mernik, M.: Unit testing for domain-specific languages. In: Taha, W.M. (ed.) Domain-Specific Languages, pp. 125–147. Springer, Berlin, Heidelberg (2009)

[11] Meyers, B., Denil, J., Dávid, I., Vangheluwe, H.: Automated testing support for reactive domain-specific modelling languages. In: Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, pp.

181–194. Association for Computing Machinery, Amsterdam, The Netherlands (2016)

[12] Khorram, F., Bousse, E., Mottu, J.-M., Sunyé, G.: Adapting TDL to provide testing support for executable DSLs. Journal of Object Technology **20**(3), 6–115 (2021)

[13] Khorram, F., Bousse, E., Mottu, J.-M., Sunyé, G.: Advanced testing and debugging support for reactive executable DSLs. Software and Systems Modeling **22**(3), 819–845 (2023) https://doi.org/10.1007/S10270-022-01025-W

[14] Cañizares, P.C., Gómez-Abajo, P., Núñez, A., Guerra, E., Lara, J.: New ideas: automated engineering of metamorphic testing environments for domain-specific languages. In: SLE '21: 14th ACM SIGPLAN International Conference on Software Language Engineering, pp. 49–54. ACM, Chicago, IL, USA (2021)

[15] Whittaker, J.A.: What is software testing? why is it so hard? practice tutorial. IEEE Softw. **17**(1), 70–79 (2000) https://doi.org/10.1109/52.819971

[16] Danglot, B., Vera-Perez, O., Yu, Z., Zaidman, A., Monperrus, M., Baudry, B.: A snowballing literature study on test amplification. Journal of Systems and Software **157**, 110398 (2019) https://doi.org/10.1016/j.jss.2019.110398

[17] Xie, T.: Augmenting automatically generated unit-test suites with regression oracle checking. In: Thomas, D. (ed.) Proceedings ECOOP 2006 - Object-Oriented Programming, 20th European Conference. Lecture Notes in Computer Science, vol. 4067, pp. 380–403. Springer, Nantes, France (2006). https://doi.org/10.1007/11785477_23

[18] Danglot, B., Vera-Pérez, O.L., Baudry, B., Monperrus, M.: Automatic Test Improvement with DSpot: a Study with Ten Mature Open-Source Projects. Empirical Software Engineering **24**(4), 1–35 (2019) https://doi.org/10.1007/s10664-019-09692-y

[19] Abdi, M., Rocha, H., Demeyer, S.: Test amplification in the pharo smalltalk ecosystem. In: Proceedings IWST 2019 International Workshop on Smalltalk Technologies. ESUG (2019)

[20] Schoofs, E., Abdi, M., Demeyer, S.: Ampyfier: Test amplification in python. CoRR **abs/2112.11155** (2021) 2112.11155

[21] Mayerhofer, T., Combemale, B.: The tool generation challenge for executable domain-specific modeling languages. In: Seidl, M., Zschaler, S. (eds.) Software Technologies: Applications and Foundations, pp. 193–199. Springer, Cham (2018)

[22] Makedonski, P., Adamis, G., Käärik, M., Kristoffersen, F., Carignani, M., Ulrich, A., Grabowski, J.: Test descriptions with ETSI TDL. Software Quality Journal **27**(2), 885–917 (2019)

[23] Coles, H., Laurent, T., Henard, C., Papadakis, M., Ventresque, A.: PIT: A practical mutation testing tool for Java (demo). In: International Symposium on Software Testing and Analysis (ISSTA), pp. 449–452. ACM, Saarbrücken, Germany (2016). https://doi.org/10.1145/2931037.2948707 . See also https://pitest.org/quickstart/mutators, https://github.com/hcoles/pitest

[24] Khorram, F., Bousse, E., Garmendia, A., Mottu, J.-M., Sunyé, G., Wimmer, M.: From coverage computation to fault localization: A generic framework for domain-specific languages. In: Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering. SLE 2022, pp. 235–248. Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3567512.3567532

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

[26] Hamlet, R.G.: Testing programs with the aid of a compiler. IEEE Transactions on Software Engineering **3**(4), 279–290 (1977) https://doi.org/10.1109/TSE.1977.231145

[27] Bousse, E., Degueule, T., Vojtisek, D., Mayerhofer, T., Deantoni, J., Combemale, B.: Execution framework of the gemoc studio (tool demo). In: Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, pp. 84–89 (2016)

[28] Khorram, F., Bousse, E., Mottu, J., Sunyé, G., Gómez-Abajo, P., Cañizares, P.C., Guerra, E., Lara, J.: Automatic test amplification for executable models. In: Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems, MODELS, pp. 109–120. ACM, Montreal, Canada (2022)

[29] Leroy, D., Bousse, E., Wimmer, M., Mayerhofer, T., Combemale, B., Schwinger, W.: Behavioral interfaces for executable dsls. Software and Systems Modeling **19**(4), 1015–1043 (2020)

[30] Steinberg, D., Budinsky, F., Merks, E., Paternostro, M.: EMF: Eclipse Modeling Framework. Pearson Education, London, UK (2008)

[31] Ammann, P., Offutt, J.: Introduction to Software Testing, 2^{nd} Edition. Cambridge University Press, Cambridge, UK (2016). https://doi.org/10.1017/CBO9780511809163

[32] Andrews, J.H., Briand, L.C., Labiche, Y., Namin, A.S.: Using mutation analysis for assessing and comparing testing coverage criteria. IEEE Transactions on Software Engineering **32**(8), 608–624 (2006) https://doi.org/10.1109/TSE.2006.83

[33] Gómez-Abajo, P., Guerra, E., Lara, J., Merayo, M.G.: Wodel-test: a model-based framework for language-independent mutation testing. Software and Systems Modeling **20**(3), 767–793 (2021) https://doi.org/10.1007/s10270-020-00827-0

[34] Harrold, M.J., Rothermel, G., Sayre, K., Wu, R., Yi, L.: An empirical investigation of the relationship between spectra differences and regression faults. Software Testing, Verification and Reliability **10**(3), 171–194 (2000)

[35] Mariani, L.: A fault taxonomy for component-based software. In: International Workshop on Test and Analysis of Component-Based Systems, TACoS 2003, Satellite Event of ETAPS 2003. Electronic Notes in Theoretical Computer Science, vol. 82, pp. 55–65. Elsevier, Warsaw, Poland (2003). https://doi.org/10.1016/S1571-0661(04)81025-9

[36] Durães, J., Madeira, H.: Emulation of software faults: A field data study and a practical approach. IEEE Trans. Software Eng. **32**(11), 849–867 (2006) https://doi.org/10.1109/TSE.2006.113

[37] Darabos, A., Pataricza, A., Varró, D.: Towards testing the implementation of graph transformations. In: Proceedings of the Fifth International Workshop on Graph Transformation and Visual Modeling Techniques, GT-VMT@ETAPS 2006. Electronic Notes in Theoretical Computer Science, vol. 211, pp. 75–85. Elsevier, Vienna, Austria (2006). https://doi.org/10.1016/J.ENTCS.2008.04.031

[38] Bartel, A., Baudry, B., Muñoz, F., Klein, J., Mouelhi, T., Le Traon, Y.: Model driven mutation applied to adaptative systems testing. In: Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Workshop Proceedings, pp. 408–413. IEEE Computer Society, Berlin, Germany (2011). https://doi.org/10.1109/ICSTW.2011.24

[39] Gómez-Abajo, P., Guerra, E., de Lara, J., Merayo, M.G.: A tool for domain-independent model mutation. Science of Computer Programming **163**, 85–92 (2018) https://doi.org/10.1016/j.scico.2018.01.008

[40] Xtext https://www.eclipse.org/Xtext/. (last accessed in February 2024) (2023)

[41] Khorram, F., Gómez-Abajo, P.: A Language-Parametric Test Amplification Framework for xDSLs: Artefacts. https://doi.org/10.5281/zenodo.7931246

[42] Pinto Ferraz Fabbri, S.C., Delamaro, M.E., Maldonado, J.C., Masiero, P.C.: Mutation analysis testing for finite state machines. In: Proceedings of 1994 IEEE International Symposium on Software Reliability Engineering, pp. 220–229 (1994)

[43] Fabbri, S.C.P.F., Maldonado, J.C., Delamaro, M.E.: Proteum/fsm: a tool to support finite state machine validation based on mutation testing. In: Proceedings. SCCC'99 XIX International Conference of the Chilean Computer Science Society, pp. 96–104 (1999)

[44] Li, J.-h., Dai, G.-x., Li, H.-h.: Mutation analysis for testing finite state machines. In: 2009 Second International Symposium on Electronic Commerce and Security, pp. 620–624 (2009)

[45] Siavashi, F., Truscan, D., Vain, J.: Vulnerability assessment of web services with model-based mutation testing. In: 2018 IEEE International Conference on Software Quality, Reliability and Security (QRS), pp. 301–312 (2018)

[46] Jeanneret, C., Glinz, M., Baudry, B.: Estimating footprints of model operations. In: 2011 33rd International Conference on Software Engineering (ICSE), pp. 601–610 (2011). IEEE

[47] Smith, B.H., Williams, L.: On guiding the augmentation of an automated test suite via mutation analysis. Empirical software engineering **14**(3), 341–369 (2009)

[48] Gómez-Abajo, P., Guerra, E., Lara, J., Merayo, M.G.: Systematic engineering of mutation operators. Journal of Object Technology **19**(3), 3–116 (2020) https://doi.org/10.5381/jot.2020.19.3.a5

[49] Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A.: Experimentation in Software Engineering. Springer, Berlin, Germany (2012)

[50] Shan, L., Zhu, H.: Generating structurally complex test cases by data mutation: A case study of testing an automated modelling tool. The Computer Journal **52**(5), 571–588 (2009) https://doi.org/10.1093/comjnl/bxm043

[51] Sun, C.-a., Liu, Y., Wang, Z., Chan, W.K.: μmt: A data mutation directed metamorphic relation acquisition methodology. In: Proceedings of the 1st International Workshop on Metamorphic Testing. MET '16, pp. 12–18. Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2896971.2896974

[52] Xuan, J., Xie, X., Monperrus, M.: Crash reproduction via test case mutation: Let existing test cases help. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. ESEC/FSE 2015, pp. 910–913. Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2786805.2803206

[53] Zhu, H.: Jfuzz: A tool for automated java unit testing based on data mutation and metamorphic testing methods. In: 2015 Second International Conference on Trustworthy Systems and Their Applications, pp. 8–15 (2015). https://doi.org/10.1109/TSA.2015.13

[54] Zeller, A., Gopinath, R., Böhme, M., Fraser, G., Holler, C.: The Fuzzing Book. https://www.fuzzingbook.org/, CISPA Helmholtz Center for Information Security (2021). Retrieved 2021-10-26 15:30:20+02:00. https://www.fuzzingbook.org/ Accessed 2021-10-26 15:30:20+02:00

[55] Tonella, P.: Evolutionary testing of classes. In: Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis. ISSTA '04, pp. 119–128. Association for Computing Machinery, New York, NY, USA (2004). https://doi.org/10.1145/1007512.1007528 . https://doi.org/10.1145/1007512.1007528

[56] Assis, T.B., Menegassi, A.A., Endo, A.T.: Amplifying tests for cross-platform apps through test patterns. In: Perkusich, A. (ed.) The 31st International Conference on Software Engineering and Knowledge Engineering, SEKE, pp. 55–74. KSI Research Inc. and Knowledge Systems Institute Graduate School, Lisbon, Portugal (2019). https://doi.org/10.18293/SEKE2019-076

[57] Allala, S.C., Sotomayor, J.P., Santiago, D., King, T.M., Clarke, P.J.: Towards transforming user requirements to test cases using MDE and NLP. In: 43rd IEEE Annual Computer Software and Applications Conference (COMPSAC), pp. 350–355. IEEE, Milwaukee, Wisconsin, USA (2019)

[58] Kriebel, S., Markthaler, M., Salman, K.S., Greifenberg, T., Hillemacher, S., Rumpe, B., Schulze, C., Wortmann, A., Orth, P., Richenhagen, J.: Improving model-based testing in

automotive software engineering. In: Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), pp. 172–180. ACM, Gothenburg, Sweden (2018)

[59] Rodriguez, J.J.G., Cuaresma, M.J.E., Risoto, M.M.: A model-driven approach for functional test case generation. Journal of Systems and Software **109**, 214–228 (2015) https://doi.org/10.1016/j.jss.2015.08.001

[60] Vieira, M., Leduc, J., Hasling, W.M., Subramanyan, R., Kazmeier, J.: Automation of GUI testing using a model-driven approach. In: Proceedings of the 2006 International Workshop on Automation of Software Test (AST), pp. 9–14. ACM, Austin, TX, USA (2006)

[61] Fröhlich, P., Link, J.: Automated test case generation from dynamic models. In: Bertino, E. (ed.) Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP). Lecture Notes in Computer Science, vol. 1850, pp. 472–492. Springer, Sophia Antipolis, France (2000)

[62] Ahmadi, R., Hili, N., Dingel, J.: Property-aware unit testing of UML-RT models in the context of MDE. In: Proceedings of the 14th European Conference on Modelling Foundations and Applications (ECMFA). Lecture Notes in Computer Science, vol. 10890, pp. 147–163. Springer, Toulouse, France (2018)

[63] Rocha, M., Simão, A., Sousa, T.: Model-based test case generation from UML sequence diagrams using extended finite state machines. Software Quality Journal **29**(3), 597–627 (2021) https://doi.org/10.1007/s11219-020-09531-0

[64] Ahmad, T., Iqbal, J., Ashraf, A., Truscan, D., Porres, I.: Model-based testing using UML activity diagrams: A systematic mapping study. Computer Science Review **33**, 98–112 (2019) https://doi.org/10.1016/j.cosrev.2019.07.001

[65] Giron, A.A., Souza Gimenes, I.M., OliveiraJr, E.: Evaluation of test case generation based on a software product line for model transformation. Journal of Computer Science **14**(1), 108–121 (2018) https://doi.org/10.3844/jcssp.2018.108.121

[66] Guerra, E., Soeken, M.: Specification-driven model transformation testing. Software and Systems Modeling **14**(2), 623–644 (2015) https://doi.org/10.1007/s10270-013-0369-x

[67] Al-Azzoni, I., Iqbal, S.: A framework for the regression testing of model-to-model transformations. e-Informatica Software Engineering Journal **15**(1), 65–84 (2021)

[68] Troya, J., Segura, S., Cortés, A.R.: Automated inference of likely metamorphic relations for model transformations. Journal of Systems and Software **136**, 188–208 (2018) https://doi.org/10.1016/j.jss.2017.05.043

[69] Bill, R., Fleck, M., Troya, J., Mayerhofer, T., Wimmer, M.: A local and global tour on momot. Software and Systems Modeling **18**(2), 1017–1046 (2019) https://doi.org/10.1007/s10270-017-0644-3

[70] Harten, N., Damasceno, C.D.N., Strüber, D.: Model-driven optimization: Generating smart mutation operators for multi-objective problems. In: 48th Euromicro Conference on Software Engineering and Advanced Applications, SEAA, pp. 390–397. IEEE, Gran Canaria, Spain (2022). https://doi.org/10.1109/SEAA56994.2022.00068

[71] Burdusel, A., Zschaler, S., John, S.: Automatic generation of atomic multiplicity-preserving search operators for search-based model engineering. Softw. Syst. Model. **20**(6), 1857–1887 (2021) https://doi.org/10.1007/S10270-021-00914-W

[72] Abdeen, H., Varró, D., Sahraoui, H.A., Nagy, A.S., Debreceni, C., Hegedüs, Á., Horváth, Á.: Multi-objective optimization in rule-based design space exploration. In: ACM/IEEE International Conference on Automated Software Engineering, ASE, pp. 289–300. ACM, New York, NY, USA (2014). https://doi.org/10.1145/2642937.2643005

[73] Strüber, D., Born, K., Gill, K.D., Groner, R., Kehrer, T., Ohrndorf, M., Tichy, M.: Henshin: A usability-focused framework for EMF

model transformation development. In: Graph Transformation - 10th International Conference, ICGT. Lecture Notes in Computer Science, vol. 10373, pp. 196–208. Springer, Marburg, Germany (2017). https://doi.org/10.1007/978-3-319-61470-0

[74] Gómez-Abajo, P., Cañizares, P.C., Núñez, A., Guerra, E., Lara, J.: Automated engineering of domain-specific metamorphic testing environments. Inf. Softw. Technol. **157**, 107164 (2023) https://doi.org/10.1016/J.INFSOF.2023.107164

[75] Semeráth, O., Varró, D.: Iterative generation of diverse models for testing specifications of DSL tools. In: Fundamental Approaches to Software Engineering, 21st International Conference, FASE. Lecture Notes in Computer Science, vol. 10802, pp. 227–245. Springer, Thessaloniki, Greece (2018). https://doi.org/10.1007/978-3-319-89363-1_13

[76] Semeráth, O., Farkas, R., Bergmann, G., Varró, D.: Diversity of graph models and graph generators in mutation testing. Int. J. Softw. Tools Technol. Transf. **22**(1), 57–78 (2020)

[77] Alhwikem, F., Paige, R.F., Rose, L., Alexander, R.: A systematic approach for designing mutation operators for MDE languages. In: Proceedings of the 13th Workshop on Model-Driven Engineering, Verification Nd Validation Co-located with ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems MODELS. CEUR Workshop Proceedings, vol. 1713, pp. 54–59. CEUR-WS.org, Saint Malo, France (2016). https://ceur-ws.org/Vol-1713