Automatic Test Amplification for Executable Models

Anonymous Author(s)

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 cases of behavioral models. In this paper, we fill this gap with an automated and generic approach. Given an executable DSL, a conforming behavioral model, and an existing test suite, the proposed approach generates 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 mutation score. We provide tool support for the approach atop the Eclipse GEMOC Studio and show its applicability in an empirical study. In the experiment, we applied the approach to 71 test suites written for models conforming to two different DSLs, and for 67 of the 71 cases, it successfully improved the mutation score between 3.17 % and 54.11 % depending on the initial setup.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; *Model-driven software engineering*.

KEYWORDS

Test Amplification, Regression Testing, Executable Model, Executable DSL

ACM Reference Format:
Anonymous Author(s). 2022. Automatic Test Amplification for Executable Models. In *ACM/IEEE 25th International Conference on Model Driven Engineering Languages and Systems, October 23–28, 2022, Montreal, Canada.* ACM, New York, NY, USA, 11 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

A vast range of Domain-Specific Languages (DSLs) are used for describing dynamic behavior of systems as *behavioral models* (e. g. state machines [**?**], activity diagrams [**?**], and process models [**? ?**]). They are used in dedicated modeling environments offering out-of-the-box tool support, including dynamic Verification and Validation (V&V) techniques that enable the user (i. e. the domain expert) to assess the correctness of the modeled behavior as early as possible [**?**]. 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 [**? ? ? ?**], while some others provide generic solutions that are applicable to a wide range of xDSLs [**? ? ? ?**]. They allow the 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 efficiently) is a difficult manual task.

In the realm of software testing, a test case generation technique called *test amplification* has recently emerged [**?**]. 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 small 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 [**? ?**].

So far, several test amplification solutions have been proposed for specific programming languages, grounded in their supporting testing frameworks (e. g. Java [**?**], Pharo Smalltalk [**?**], and Python [**?**]). 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 [**? ? ? ?**] and the engineering of new ones is recurrent [**?**], developing a test amplification approach for each and every xDSL is costly and potentially repetitive.

In this paper we propose a generic and automated approach for amplifying test suites of executable models. We focus on regression testing as a goal for test case generation, and we use as a starting point the testing framework proposed by Khorram et al. [**?**], which supports the definition and execution of test cases for any executable model using the standard Test Description Language (TDL) [**?**]. 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 the modifiers suggested for JUnit test cases [**? ?**], 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 the resulting execution trace is captured. From this trace, all the possible assertions for

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MODELS, October 23–28, 2022, Montreal, Canada
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06...$15.00
https://doi.org/XXXXXXX.XXXXXXX

the new test cases are generated. Third, mutation analysis [? ?] is performed to select only those new test cases that improve the mutation score of the original test case. This ensures that only the most efficient test cases are proposed to the domain expert.

We implemented the proposed framework for the GEMOC Studio [?], a language and modeling workbench for xDSLs. We ran an experimentation with 71 test suites written for 71 models conforming to two different xDSLs, which successfully generated 244 new test cases improving the original mutation scores. A mutation analysis shows that on 12.481 mutants generated for the 71 models, the amplification approach improved the mutation score for 67 of the 71 test suites, ranging from 3.17 % to 54.11 % depending on the initial setup. This result reveals the effectiveness of test amplification in the context of executable model testing.

Paper organization. Section 2 provides the background and a running example. The proposed approach is then introduced in Section 3 and its supporting tool is explained in Section 4. Section 5 presents the evaluation of our approach. Finally, the related work is provided in Section 6 and Section 7 concludes the paper.

2 BACKGROUND

This section introduces a running example that will be used across the paper (Section 2.1), provides a definition of executable DSLs (Section 2.2) and their testing support (Section 2.3), and a definition of test amplification (Section 2.4). The section finishes with a discussion of the motivation and the objectives of this paper (Section 2.5).

2.1 Running Example: Arduino

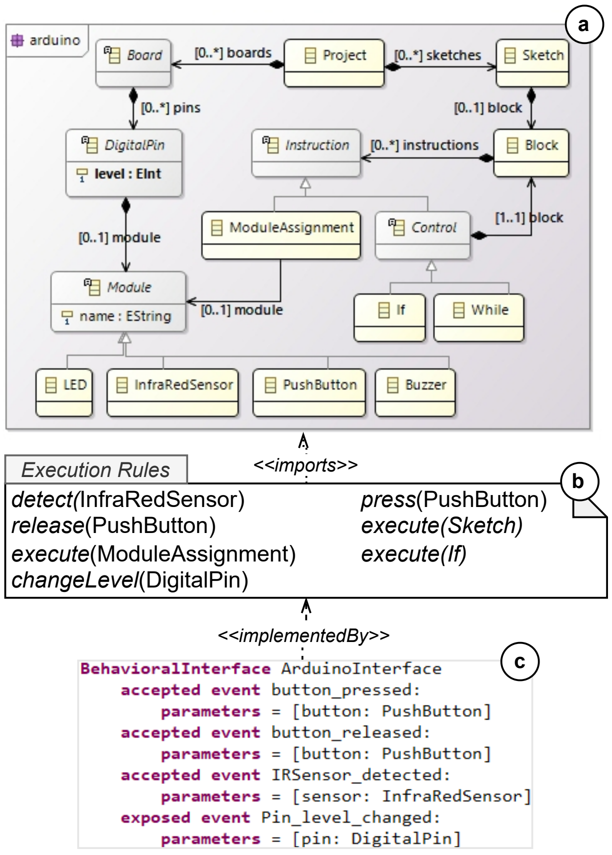
Arduino[1] is an open-source company that offers hardware boards with embedded CPUs, and with different modules (e. g. sensors, LEDs, actuators) that can be attached to a board. An Integrated Development Environment (IDE) is available to develop programs (called sketches) for such boards in C or C++. 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 [?]; and (iii) a behavioral interface defining how to interact with a running model [?].

2.2.1 Abstract Syntax. We consider the abstract syntax of an xDSL to be defined by an Ecore metamodel [?]. A metamodel is generally 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. The class Project is its root element and may contain several Board and Sketch elements. A Board represents an Arduino 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 to define conditional behaviors (e. g. using If or While).

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



```
BehavioralInterface ArduinoInterface
    accepted event button_pressed:
        parameters = [button: PushButton]
    accepted event button_released:
        parameters = [button: PushButton]
    accepted event IRSensor_detected:
        parameters = [sensor: InfraRedSensor]
    exposed event Pin_level_changed:
        parameters = [pin: DigitalPin]
```

Figure 1: An excerpt of an Arduino xDSL (called xArduino)

Figure 2 shows an example xArduino model in 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".

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.

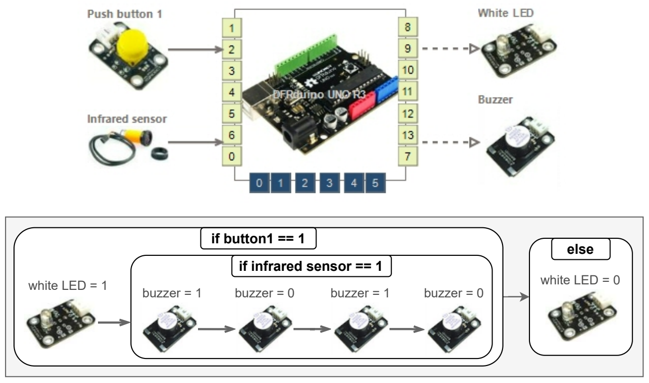


Figure 2: An example xArduino model

We assume 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 a 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 is able to interact with its environment through *events*, and must be implemented by the execution rules of the operational semantics [**?**]. Figure 1(**c**) presents a behavioral interface for xArduino. It comprises a set of events, each containing a set of parameters conforming to the xDSL's abstract syntax. Considering a running model, accepted events indicate 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. Please note that this paper assumes the accepted events are processed synchronously–often referred to as 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 [**?**], Khorram et al. proposed a generic testing framework for xDSLs using the standard Test Description Language (TDL) [**?**]. Given an xDSL, the

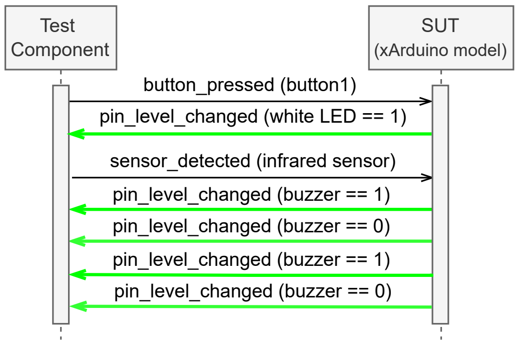

Test Component
SUT (xArduino model)
button_pressed (button1)
pin_level_changed (white LED == 1)
sensor_detected (infrared sensor)
pin_level_changed (buzzer == 1)
pin_level_changed (buzzer == 0)
pin_level_changed (buzzer == 1)
pin_level_changed (buzzer == 0)

Figure 3: A TDL test case for the xArduino model of Figure 2

framework generates an xDSL-specific TDL library that provides facilities to define test cases and test data for the conforming models based on the xDSL definition, and then executing them and obtaining the result. However, they do not support xDSLs with behavioral interfaces, meaning that it is not 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 this paper we combine the testing approach of [**?**] with the behavioral interface notation [**?**] to enable writing TDL test cases for executable models as follows: defining test input data and expected output (i. e. used in the assertions) 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 sender is the test component, the exchanged message carries test input data i. e., an accepted event instantiated from the behavioral interface of the xArduino DSL. Messages from the SUT are assertions and their carried data is the expected output i. e., the exposed event instances. The values of the events' parameters are elements of the xArduino model with values of their runtime features (such as `white LED` 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 in the bottom part of the xArduino model from Figure 2, except for the `else` part. As shown by the green arrows in Figure 3, the test case is a success.

2.4 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 [**?**]. A subset of these techniques is focused on improving manually-written test cases to avoid regression faults. Given a test suite for a system, such techniques create new test cases by modifying the test input data of existing 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

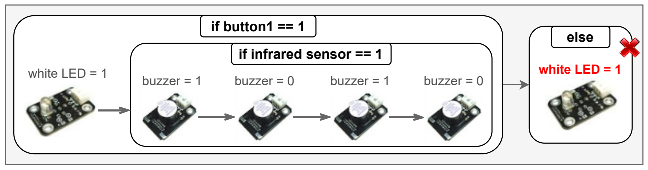


Figure 4: Part of a mutant generated for the xArduino model of Figure 2 by changing the assignment of the `else` part

test cases are based on the current behavior of the system, these techniques can effectively strengthen regression testing [? ?].

As test amplification techniques are able to generate very large amounts of new test cases, an important additional step is to filter the resulting test suite to only keep relevant test cases. When the goal is to increase the effectiveness of test cases in detecting regression faults, an efficient technique for identifying relevant test cases is mutation analysis [?]. This technique can inject artificial faults in the SUT, and then measures the degree to which an existing test case detects such faults. Artificial faults are modeled via sets of *mutation operators*, which perform small modifications (e.g. flipping '>' by '<' in an expression, or changing the value of constants) on the source code. These operators are systematically applied to the SUT to produce a set of *mutants*. For example, Figure 4 shows part of a mutant for the xArduino model of Figure 2 that is generated by changing the value used in the assignment of the `else` part.

Mutants can be used to evaluate the quality of the test cases. If the original program and the mutant produce different outputs, then the test suite 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 xArduino mutant (Figure 4), the output will be the same as running it on the original xArduino model (Figure 2). Thus, this test case is not able to kill the mutant. The mutation score, which is the percentage of killed mutants, provides a measure of the test suite quality [?].

In the proposed approach, we use mutation analysis to check the degree of improvement that test amplification provides, and as a selection mechanism for the more effective amplified test cases.

2.5 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 to the system under test and generating new assertions from the system execution traces. Both tasks require working with system-specific data which could shape very differently depending on the language the system is implemented with. Consequently, all the existing test amplification solutions are dedicated to specific programming languages and are grounded on their supporting testing frameworks (e. g. Java [?], Pharo Smalltalk [?], and Python [?]).

In the context of this paper, the system under test is an executable model defined by an xDSL. As there are plenty of xDSLs for designing systems of specific domains [? ? ? ?] and as engineering new ones for emerging domains is recurrent [?], there is a strong incentive to provide a *generic* test amplification approach applicable to any xDSL. Such an approach must be adaptable for every given xDSL since each one of them has its own definition of data, which could highly differ from one to another. It must also be founded on a generic testing framework that supports the same kind of xDSLs (according to the definitions given in Section 2.2).

In the next section, we propose a generic test amplification approach for xDSLs. We base it on the testing framework proposed by Khorram et al. [?] because it supports the xDSLs considered in the context of this paper.

3 APPROACH

In this section, we first present an overview of the proposed approach (Section 3.1) and then detail its main components (Sections 3.2–3.4).

3.1 Overview

Figure 5 shows an overview of the proposed approach, which involves two roles. First, a language engineer (on the top center) who defines an xDSL according to the definitions given in Section 2.2. Additionally, we assume the availability of a set of mutation operators for the xDSL. Second, a domain expert (on the top left) who defines a model (using the xDSL) and a TDL test suite for them.

The first component of the approach is the *Test Case Modifier* (on the left side). It takes the manually-written test suite and the model under test, and generates new test cases by modifying the given test suite (label 1). This modification involves changing the test input data and removing the existing assertions since they are no longer valid due to the data changes. As the test input data are specific to the xDSL that the tested model conforms to, this component uses the xDSL definition for performing the modification. The new test cases are then given to the *Test Runner* (on the bottom left) which is the

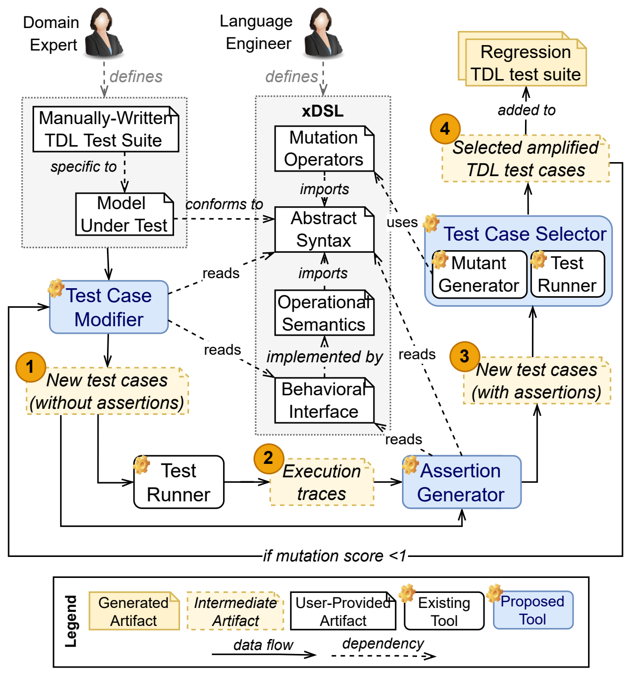


Figure 5: Approach Overview

TDL test execution engine proposed in [?]. The engine executes the test cases on the model and produces their execution traces (label 2).

Our second component is called *Assertion Generator* (on the bottom right) and follows the idea of regression oracle checking [?]. In this technique, assertions are generated based on the execution traces to improve the strength of the regression testing. Accordingly, after running each new test case, our proposed component 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. Again, as the execution trace comprises data conforming to the xDSL that the model conforms to, this component also uses the abstract syntax and the behavioral interface of the xDSL definition, this time for generating assertions. At the end, its output is a set of new test cases with assertions for regression testing (label 3).

The *Test Case Selector* is the third component of the approach (on the right side). It is in charge of filtering the generated test cases based on some given criteria. Currently, we consider the ability of new test cases in improving the fault localization capability. Accordingly, this component uses an existing mutant generator that produces mutants out of the model under test if the language engineer provides a set of mutation operators for the xDSL (on the top center). It then runs the new test cases on the mutants (using the test runner) and keeps those improving the initial mutation score (i. e. the mutation score of the given manually-written TDL test suite). The process can be iterated on the selected new test cases based on some stop criteria such as reaching a 100 % mutation score (label 4).

3.2 Test Case Modification

The first step of our test amplification process is the modification of existing test cases. 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 call *modifier* an operator that, when applied to a specific element of an existing test case, generates a new test case that is identical to the former one but for a single modification. A modifier can be applied multiple times on the same test case, yielding a different result depending on the chosen element of the test case. As some modifiers may produce too many different new test cases from a single test case, each modifier may possess its own *application policy*, which tells on which elements and how many times the modifier will be applied on each test case.

In the proposed approach, we use the following sets of modifiers.

3.2.1 Modification of Primitive Data. We adapt two existing sets of modifiers for modifying test input data that comprise primitive values: (1) the operators proposed by Danglot et al. in [?] for the amplification of JUnit test cases, and (2) the modifiers used by the Pitest mutation testing tool [?] for putting Java programs in new runtime states. The resulting modifiers as follows:

- A *numeric* value n is replaced by either 1, -1, 0, $n+1$, $n-1$, $n \times 2$, $n \div 2$, 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, or replacing the string with a random string of the same size.
- A *boolean* value is negated.

Each of these modifiers is applied as many times as possible on each considered test case. For instance, the integer modifier will be applied 16 times on a test case containing an event occurrence with two integer parameter (2×8 possibilities).

3.2.2 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 Duplication*: Duplicate an existing event occurrence. Applied on each possible event occurrence of the test case.
- *Event Deletion*: Removes an event occurrence. Applied on each possible event occurrence of the test case.
- *Event Permutation*: Performs a random permutation of two input event occurrences to generate a new test case.
- *Event Creation*: Creates a new event occurrence in the test case. First, the available accepted events of the xDSL's behavioral interface that are not used in the test input data of the given test case are collected. The availability is verified by analyzing whether for the unused event occurrences, a value can be set to their parameters using information from the model under test. If possible, this operator adds new event occurrences to the input event sequences by creating all the possible instances of the available accepted events. In particular, it generates one new test case per event addition as well as a new test case containing all the new instantiated event occurrences. In the later case, when several new event occurrences are added, this new input is modified using other operators such as event duplication, event deletion, and event permutation to generate more new test cases.
- *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. With each possible modification, it generates a new test case.

The output of this step is a set of new test cases but still without any assertion. As mentioned in Section 3.1, our approach requires execution traces to generate regression assertions. Accordingly, we execute each new test case on the model under test using the TDL test runner [?] to capture an execution trace (label 2 in Figure 5).

3.3 Assertion Generation

As earlier 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 that the model conforms to. On the other hand, as the test cases are implemented in TDL, the assertions must be generated in TDL as well. Therefore, the main role of the assertion generator component is to transform the exposed event instances to TDL elements. This transformation uses the definition of both the behavioral interface and the abstract

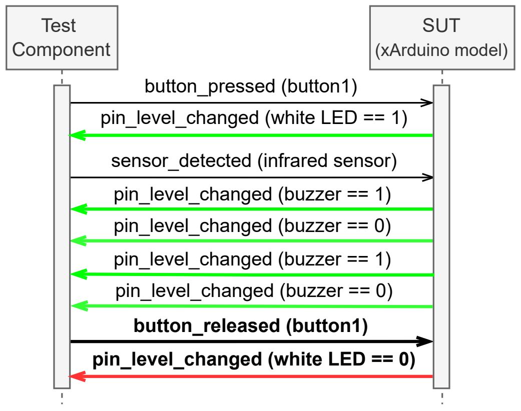

Test Component
SUT (xArduino model)
button_pressed (button1)
pin_level_changed (white LED == 1)
sensor_detected (infrared sensor)
pin_level_changed (buzzer == 1)
pin_level_changed (buzzer == 0)
pin_level_changed (buzzer == 1)
pin_level_changed (buzzer == 0)
button_released (button1)
pin_level_changed (white LED == 0)

Figure 6: An amplified TDL test case generated from the initial test case of Figure 3 by applying the *event creation* modifier. The last message unsatisfies the assertion then the test case fails, killing the mutant of Figure 4 and improving the initial mutation score

syntax of the xDSL because exposed event instances conform to the behavioral interface and they may carry EObjects of the model under test, which conform to the abstract syntax metamodel.

For example, Figure 6 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 (white LED == 0)` is generated because according to Figure 2, by releasing the button the `else` part of the Sketch will be executed which changes the level of the `white LED` to 0.

3.4 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 test suite. In this step, we rely on mutation analysis to evaluate whether a generated test case improves the quality of the test suite. Figure 7 shows the test case selection process. 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 killed and alive mutants.

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 new killed mutants to the existing set of killed mutants, increasing the mutation score (step 6), and iterates the process on the selected amplified test cases (step 7). The process finishes based on some given stop criteria. Currently, we iterate up to three times while the mutation score is less than 100 %.

For example, the new test case of Figure 6 executed on the xArduino mutant (Figure 4) fails. This means that the test case is able

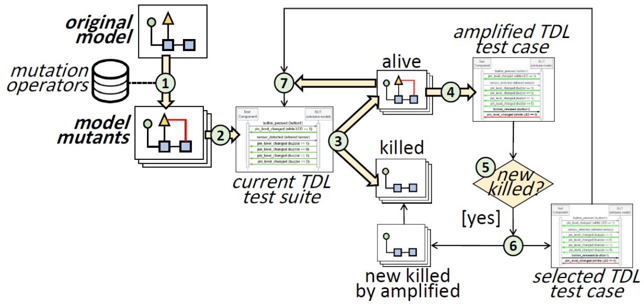


Figure 7: Selecting test cases using mutation analysis

to detect the injected fault, and therefore that the mutant has been killed, which increases the mutation score. Hence, the test case is added to the regression TDL test suite of the xArduino model.

4 TOOL SUPPORT

We implemented our proposed approach as part of the GEMOC studio [?], a language and modeling workbench for xDSLs defined on top of the Eclipse Modeling Framework (EMF) [?]. We conveniently used the testing framework proposed by Khorram et al. [?] as it is also part of the GEMOC studio and provides support for TDL, as explained in Section 2.3.

For the test case selection using mutation analysis, we exploit WODEL-Test [?], a tool that is also based on EMF technologies. This tool offers a dedicated DSL that can be used to define sets of mutation operators for any DSL defined by an Ecore metamodel. The tool helps in evaluating the coverage of such set for the language by providing coverage metrics of the operators w. r .t the metamodel [?]. Once the operator set is designed, the tool can systematically apply those operators on initial models, to create sets of mutants. The tool can also perform mutation analysis, applying test cases on the sets of mutants, and calculating the mutation score.

Overall, using the provided tool, a domain expert can select an existing TDL test suite and run the amplification process on it. This generates two files: one containing the new test cases and another reporting their mutation analysis results including the initial mutation score, the final mutation score, the mutants killed by each of them, and the mutants that are still alive despite the amplification. The source code is available on a GitLab server[3].

5 EVALUATION

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

RQ1 How much genericity is provided by the framework in terms of the supported xDSLs?

RQ2 To what extent do the generated test cases increase the mutation 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?

[3] Unfortunately, we are not able to share our code with the reviewers as it would most probably reveal our names, due to the used acronyms, package names or code comment, and of course, we do not want to break the double-anonymous review process. In case the paper gets accepted, we will publish all our code in a public repository.

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
	#mutation operators	36	28
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
	#generated mutants	394	12,087

In the following, we describe the experiment setup (Section 5.1), answer the RQs (Section 5.2) and describe threats to validity of the experiments (Section 5.3). The evaluation data is accessible from an anonymous git repository[4].

5.1 Experiment Setup

Setup for RQ1. For the first research question, we intend to evaluate the applicability of our approach to two different xDSLs. The first xDSL is xArduino, already presented in Section 2.2. The second is xDSL 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) [?]. 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.

As our approach relies on mutation analysis, we defined a set of mutation operators for each xDSL using the WODEL framework [?]. We created 36 operators for xArduino, which only mutate the behavior part of the models, ignoring the physical-related concepts. For xPSSM, we defined 28 mutation operators, based on previous work on state machine mutation [? ? ? ? ?]. Overall, the metamodel footprint [?] of the mutation operators covers 69.56 % of the xArduino metamodel and 50 % of the xPSSM metamodel.

For each xDSL, we prepared a set of models of different sizes as presented 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 48 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 [?]. From this existing 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 ranging between 13 and 69 objects. Hence, overall we considered a total of 65 xPSSM models (5+60).

Setup for RQ2. The second research question aims to assess the ability of the proposed approach for improving manually-written test cases. For this, we manually wrote a set of TDL test cases for each considered model. Cumulatively, we wrote 22 test cases for the xArduino models and 216 test cases for the xPSSM models, where 60 of them were the TDL versions of the test cases provided by the standard PSSM test suite [?].

As explained in Section 3, our approach relies on mutation analysis to measure the degree of achieved improvement on the test suite quality. Accordingly, we used the WODEL mutant generator [?] 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.

Setup for RQ3. The third research question aims to investigate whether the *size* and the *quality* of the initial test suite impacts the amplification result. In this regard, we classified our provided TDL test suites into 4 categories:

(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 $\geq 80\,\%$

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 [?].

For every category and xDSL, Table 2 presents the number of original test cases (column 3), the number of generated mutants (column 4), the number of mutants killed by the original test cases (column 5), and the original mutation score (column 6). The provided numbers are cumulative numbers, and the scores are the average scores considering all provided models and test suites. It should be noted that, when possible, we intentionally reduced the mutation score of a MSHQ test suite to have a new version of it as MSMQ or SSMQ (i. e. if reducing the mutation score had resulted in keeping only one test case in the new version of the test suite).

5.2 Evaluation Result

Answering RQ1. The purpose of the first research question is to assess whether the approach can amplify test cases for various xDSLs. To answer this question, we used the prototype presented in Section 4 for the two considered xDSLs, and executed the test amplification tool on the 71 test suites. For 67 of them, new test cases were successfully generated—a total of 244—improving their original mutation score between 3.17 % and 54.11 % on average, based on the initial setup (detailed results are given shortly after while answering the next research questions). Therefore, we can conclude that the approach does provide a certain level of genericity, i. e. the approach is not solely dedicated to a single specific xDSL, and can be applied on at least two different ones.

It is also worth mentioning that, to support an additional xDSL with our test amplification approach, the only additional cost for the language engineer is to define a set of mutation operators for the xDSL. These operators are defined once and can be reused for other purposes, such as test quality measurement [? ?] and fault localization based on mutation analysis [?].

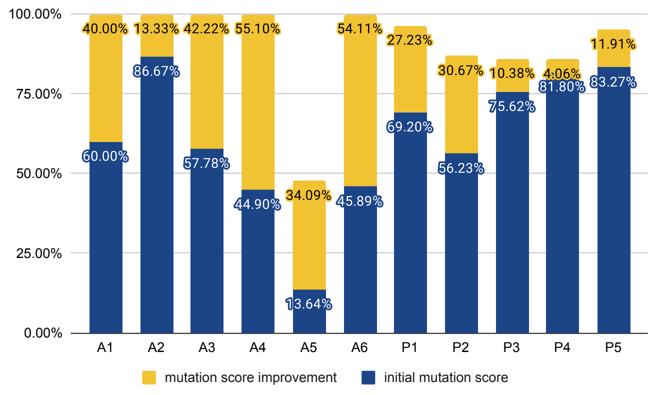
[4] https://anonymous.4open.science/r/AutoTestAmplificationForModels-16EC

Table 2: Evaluation result for amplifying test suites with different sizes and qualities

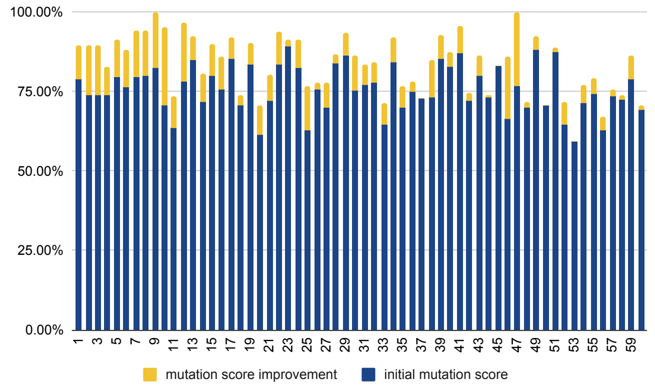
		# Orig. test cases	# Generated mutants	# Killed mutants by orig. test cases	Orig. mutation score	# New ampl. test cases	# New killed mutants by ampl. tests	Mutation score improvement	Regression test suite size (# orig. + # ampl.)	Regression test suite global score (orig. + improv.)
SSMQ	xArduino	4	148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

Answering RQ2. To answer the second research question, we must evaluate the degree of improvement provided by the generated test cases. Figure 8 presents the results for the selected 71 test suites using bar charts. Figure 8(a) displays the result for the 11 test suites of manually defined models, that is, 6 xArduino models (A bars) and 5 xPSSM models (P bars). We have obtained mutation score improvements for all test suites, ranging from 4.06 % (P4) to 55.10 % (A4). Additionally, the final mutation score for 5 test suites (A1-A4 and A6) reaches 100 % after amplification. Figure 8(b) shows the result for the 60 test suites from the PSSM standard [**?**]; where each test suite has only one test case. 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). This also means that, even when starting from small test suites with just one test case, the approach is able to provide improvement. These results reveal the success of our test amplification approach in improving the mutation score of the models' test suites. However, the rate of improvement is different case by case, as we will discuss while answering RQ3.

Answering RQ3. Given a test suite to be amplified, its *size* (i. e. number of test cases) and *quality* (i. e. 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 in four different setups. 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 7) and the average improvement (column 9) for high-quality tests is less than the one of 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 %, but for the SSHQ ones, it is improved from 86.67 % to 100 %. 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



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



(b) Mutation score improvement for existing data set

Figure 8: Mutation score improvement by test amplification

test suites improves from 71.99 % to 80.68 %, and for the SSHQ ones from 84.38 % to 91.51 %. This could imply that, when the original test cases have higher quality, there is more room for test amplification. By amplifying high-quality tests, it is more probable to generate new *effective* test cases.

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. 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. Hence, the more available initial test cases, the more generated test cases, and the more chances for improving of the test suite quality.

5.3 Threats to Validity

We listed multiple threats to the validity of our experiment. Although we tried to apply the approach on two xDSLs from very different domains, there is still an external threat on the genericity of our approach. Therefore, we would like to apply our approach on 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, we plan to explore the approach on xDSLs with more complex behavioral interfaces.

We iterated the amplification process while there is a 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 mutation score (label 4 in Figure 5). Without this test case selection criterion (i. e. iterating from label 3 of Figure 5), the number of generated test cases increases exponentially after a few iterations. However, the non-effective test cases (i. e. test cases not contributing in improving the mutation score in the current iteration) might become effective in the next iterations since a combination of several modifiers are applied on them. It is also worth mentioning that we experimented the tool for up to 3 iterations while the mutation score is less than 100 % to avoid a huge experimentation time; nonetheless, we may reach higher mutation scores with more iterations. Hence, the users of the tool are allowed to change this stop criterion.

For test case modification, we used the modifiers presented in Section 3.2. Other more complex modifiers could be devised, which may show more effectiveness in fault localization. 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.

As discussed in Section 3, we assume that a set of mutation operators has been provided for each considered xDSL. Depending on the quality of these operators (e. g. their metamodel footprint) the diversity of the generated mutants would differ, leading to variations in the effectiveness of the test amplification tool. It would be interesting to consider other test case selection criteria such as increasing coverage, which is left to our future work.

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.

6 RELATED WORK

This section provides an overview of related research regarding test input data modification (Section 6.1), test amplification for regression testing (Section 6.2) and test case generation for behavioral models (Section 6.3). In the three areas, we identify the innovations of our approach.

6.1 Test Input Data Modification

Data mutation testing [**?**] 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 [**? ? ?**]. Sun et al. [**?**] 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 [**?**] 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. [**?**] 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 [**?**], which consists of generating random input data as a test case, and monitor 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.

6.2 Test Amplification for Regression Testing

Several approaches use test amplification for regression testing. Xie [**?**] 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 suites by systematically adding assertions for improving their capability of avoiding regression faults. DSpot [**?**] targets the automatic amplification of JUnit test cases. It combines input space exploration [**?**] with regression oracle generation [**?**] 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 which improve the mutation score. On the basis of DSpot, Abdi et al. [**?**] propose Small-Amp, an amplification approach for the Pharo Smalltalk ecosystem. Ebert et al. [**?**][**?**] 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. [**?**] 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 system under test. However, the test input modifiers are defined using a set of test patterns specific to the context of mobile applications.

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

6.3 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 [**? ? ? ?**]. Most of these efforts follow one of two main approaches for test case generation: path/coverage analysis [**? ?**], or category partition [**? ?**]. 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 [**?**] 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 [**?**] present an approach for 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. [**?**] 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. [**?**] generate test cases with input data to cover all executable paths of the diagrams, together with their expected output. The interested reader can consult [**?**] 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. [**?**] use software product lines and input metamodel coverage to generate a reduced set of test cases for model transformations; Guerra and Soeken [**?**] use constraint solving to generate test models and partial oracles from declarative transformation specifications; Al-Azzoni and Iqbal [**?**] apply test case prioritization for regression of transformations based on an analysis of the transformation rules' coverage; and Troya et al. [**?**] infer likely metamorphic relations for ATL model 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 (test case amplification) are different.

Innovation of our approach. Altogether, we find a variety of approaches for test case generation in the modeling area, but to the best of our knowledge, ours is the first proposal targeting test case improvement for xDSLs. We believe that 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.

7 CONCLUSION AND FUTURE WORK

In this paper, we have presented an approach for amplifying test cases of executable models. The method targets the regression testing of models built using xDSLs, where TDL is used to describe test cases. The method is generic and applicable to any xDSL. We propose tool support atop the GEMOC studio and report on an evaluation on two different xDSLs that shows the benefits of the method in terms of the improved effectiveness of the amplified test cases, as given by the mutation score.

As future work, we would like to expand our tool support. For example, input modifiers changing event sequences may need to create dynamic objects following certain criteria. We envision using search-based techniques, e. g. based on MOMoT [**?**] for this purpose. Moreover, the whole amplification process could be recast as a search process, e. g. based on genetic algorithms. This way, 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 mutation score improvement. We would also like to make explicit and extensible the set of input modifiers, possibly via a dedicated DSL. Another interesting extension would be to consider coverage as a criterion for our test case selector component. With regards to the evaluation, we would like to expand our experiments to be able to answer other interesting research questions, such as which are the most effective modifiers, or which kind of mutants are killed by each used modifier. Finally, given the promising results obtained so far, we aim at investigating test case amplification for other purposes than regression testing.