From Coverage Computation to Fault Localization: A Generic Framework for Domain-Specific Languages

Faezeh Khorram, Erwan Bousse, Antonio Garmendia, Jean-Marie Mottu, Gerson Sunyé, Manuel Wimmer

From Coverage Computation to Fault Localization: A Generic Framework for Domain-Specific Languages

Faezeh Khorram
Erwan Bousse
faezeh.khorram@imt-atlantique.fr
erwan.bousse@ls2n.fr
IMT Atlantique, Nantes Université, École Centrale Nantes, CNRS, LS2N, UMR 6004, F-44000
Nantes, France

Antonio Garmendia
antonio.garmendia@jku.at
Institute of Business Informatics - Software Engineering
Johannes Kepler University Linz
Linz, Austria

Jean-Marie Mottu
Gerson Sunyé
jean-marie.mottu@ls2n.fr
gerson.sunye@ls2n.fr
IMT Atlantique, Nantes Université, École Centrale Nantes, CNRS, LS2N, UMR 6004, F-44000
Nantes, France

Manuel Wimmer
manuel.wimmer@jku.at
CDL-MINT, Institute of Business Informatics - Software Engineering
Johannes Kepler University Linz
Linz, Austria

Abstract

To test a system efficiently, we need to know how good are the defined test cases and to localize detected faults in the system. Measuring test coverage can address both concerns as it is a popular metric for test quality evaluation and, at the same time, is the foundation of advanced fault localization techniques. However, for Domain-Specific Languages (DSLs), coverage metrics and associated tools are usually manually defined for each DSL representing costly, error-prone, and non-reusable work.

To address this problem, we propose a generic coverage computation and fault localization framework for DSLs. Considering a test suite executed on a model conforming to a DSL, we compute a coverage matrix based on three ingredients: the DSL specification, the coverage rules, and the model's execution trace. Using the test execution result and the computed coverage matrix, the framework calculates the suspiciousness-based ranking of the model's elements based on existing spectrum-based techniques to help the user in localizing the model's faults. We provide a tool atop the Eclipse GEMOC Studio and evaluate our approach using four different DSLs, with 297 test cases for 21 models in total. Results show that we can successfully create meaningful coverage matrices for all investigated DSLs and models. The applied fault localization techniques are capable of identifying the defects injected in the models based on the provided coverage measurements, thus demonstrating the usefulness of the automatically computed measurements.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Keywords: Testing, Coverage, Fault Localization, Executable Domain-Specific Languages, Executable Models

ACM Reference Format:
Faezeh Khorram, Erwan Bousse, Antonio Garmendia, Jean-Marie Mottu, Gerson Sunyé, and Manuel Wimmer. 2022. 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 '22), December 06–07, 2022, Auckland, New Zealand*. ACM, New York, NY, USA, 14 pages. https://doi.org/XXXXXXX.XXXXXXX

SLE '22, December 06–07, 2022, Auckland, New Zealand

ACM ISBN 978-1-4503-XXXX-X/22/12…$15.00
https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

A multitude of Domain-Specific Languages (DSLs) are available nowadays to describe the dynamic aspects of systems as *behavioral models* (e.g., see state machines, activity diagrams, and process models [5, 39, 41]). For such DSLs, the environment should not only support creating and executing behavioral models, but also dynamic Verification and Validation (V&V) techniques to assess as early as possible their correctness [22]. As these techniques need model execution, we focus in this paper on DSLs with operational semantics, referred to as *executable DSLs (xDSLs)*.

A prominent dynamic V&V technique is *testing*, which involves executing systems and observing whether they act as

expected. Some testing approaches are already proposed for xDSLs which allow defining test suites for executable models. Among them, some are tailored for a specific xDSL [23, 29, 32, 35] and some others provide generic solutions to be compatible with a wide range of xDSLs [14, 26, 27, 34, 50].

To test models efficiently, we need both to evaluate the quality of the defined test cases, and to localize the model's faults when test cases fail. Both of these concerns can be addressed by measuring the *coverage* of each executed test case, i.e., the model elements involved in the test case execution. In the realm of programming languages, both coverage metrics [1] and coverage-based fault localization techniques, such as Spectrum-Based Fault Localization (SBFL) [49], have existed for a long time. However, to our knowledge, these concerns are still understudied when it comes to xDSLs, for which coverage tools still have to be manually developed.

In this paper, we propose a generic framework for coverage computation and fault localization of domain-specific models which is applicable to a wide range of xDSLs. Considering a test suite for an executable model, we analyze the model's execution traces to extract its covered elements which compose the coverage matrix for the test suite. In addition, our proposed framework allows language engineers to customize the generic coverage measurements for their xDSLs. Finally, we investigate the application of the computed coverage measurements for fault localization in executable models based on SBFL techniques. We reuse an existing collection of SBFL techniques from the literature [45] for calculating the suspiciousness-based ranking of elements of executable models.

The proposed framework is implemented for the GEMOC Studio [9] that is a generic language and modeling workbench for xDSLs. We conducted an empirical evaluation of our approach for four different xDSLs to assess its applicability. In total, we wrote 297 test cases for 21 executable models with sizes ranging from 7 to 571 elements. We injected faults into these executable models using a model mutation tool [21] and we executed our approach for 1252 mutants of the executable models. We observed that meaningful coverage matrices can be automatically constructed for the test suites of all examined mutants and that it allows the application of existing SBFL techniques for successfully tracking the faulty model elements, thus demonstrating the usefulness of the generically computed coverage measurements.

Paper organization. We provide the background and a running example in Section 2. Then we introduce our approach in Section 3 and its supporting tool in Section 4. Section 5 presents the evaluation of our approach. Finally, related work is given in Section 6 and Section 7 concludes the paper.

2 Background

In this section, we present the background and a running example that will be used throughout the paper. At the end, we discuss the motivation and the objectives of this paper.

2.1 Running Example: Arduino

Arduino[1] is an open-source company proposing hardware boards with embedded CPUs and an Integrated Development Environment (IDE) to develop programs for the boards in C or C++. However, an xDSL specifically defined for Arduino would help in developing the Arduino programs using required concepts rather than technical C instructions. In addition, such xDSL enables the simulation and the validation of the boards before deployment. In subsequent, we present the definition of an Arduino xDSL along with its usage.

2.2 Executable Domain-Specific Languages

A Domain Specific Language (DSL) with an operational semantics is commonly referred to as an *executable DSL (xDSL)*. We call *executable model*, or xModel, a model that conforms to an xDSL. An xModel is essentially a program written using an xDSL, and which can be executed according to the xDSL semantics. Additionally, the person who defines an xDSL and the language user are often called *language engineer* and *domain expert*, respectively.

An xDSL is at least composed of three parts: (*i*) an abstract syntax specifying the domain concepts of the xDSL and the relationships between them; (*ii*) an operational semantics enabling the execution of the xModels; and (*iii*) one or several behavioral interfaces each specifying how to interact with a running xModel. In this paper, we use the GEMOC Studio to specify and tool up xDSLs [9], but these considerations also apply to other xDSL engineering platforms.

2.2.1 Abstract Syntax. Figure 1(**a**) presents an excerpt of the abstract syntax definition of an Arduino xDSL[2] using the Ecore language [44] which is similar to the core of UML Class Diagrams. The root element is a Project which may contain several Board and Sketch elements. A Board represents an Arduino physical board. It contains several DigitalPin each associated with one Module such as LED, InfraRedSensor, PushButton, and Alarm. The DigitalPin has a level 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.

The intended behavior of the boards must be defined using Sketch elements. A Sketch may contain a Block that may comprise several Instructions such as ModuleAssignment for changing the state of a Module, Delay for waiting a specific amount of time, 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

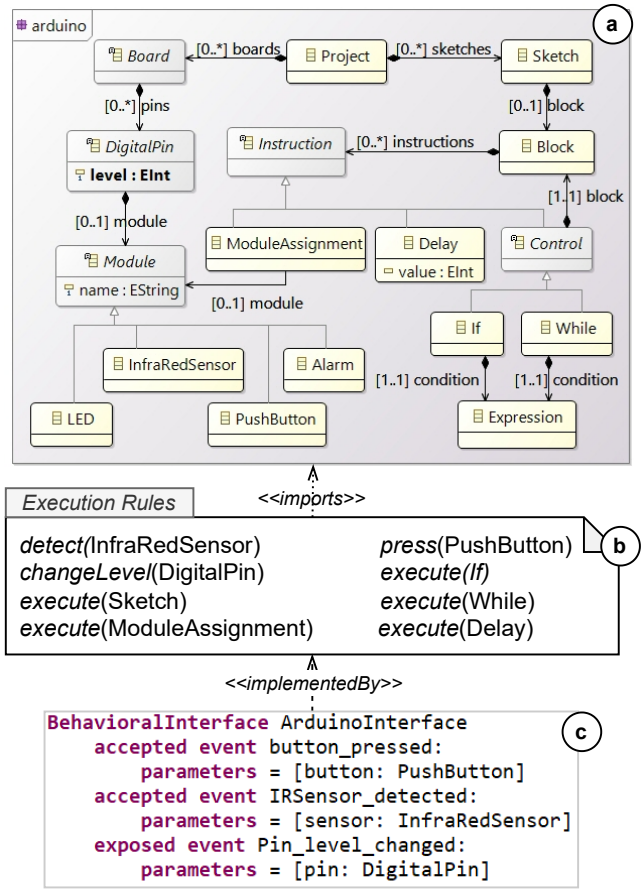

Figure 1. An excerpt of an xDSL defined for Arduino

Figure 2 shows a sample Arduino xModel, defined by instantiating the abstract syntax elements. At the top, there is an Arduino Board with 14 DigitalPins, four of them connected to Module instances: a PushButton (`button1`), an Alarm (`alarm1`), an LED (`LED1`), and an InfraRedSensor (`irSensor1`). The behavior of the board is defined in a Sketch instance (shown in the bottom of Figure 2) as: "if `button1` is pressed, `LED1` turns on (i.e., activating the alarm system), and then if `irSensor1` detects an obstacle, `alarm1` alternates between noise/silence with a one-second delay (i.e., reporting an intrusion)". Note that (*i*) pressing a button and sensing an obstacle by a sensor means the level of their DigitalPins is equal to 1; and (*ii*) turning on/off an LED and an alarm—'LED1=1', 'alarm1=1', 'LED1=0', and 'alarm1=0' notations in Figure 2—are ModuleAssignment instances. We intentionally inject a defect in this model where `alarm1` should be set to 0 but it is mistakenly set to 1, meaning that the alarm turns on but does not alternate between noise/silence states (highlighted in red in Figure 2).

2.2.2 Operational Semantics. An xDSL's operational semantics defines the possible runtime states of an xModel under execution as well as a set of execution rules specifying

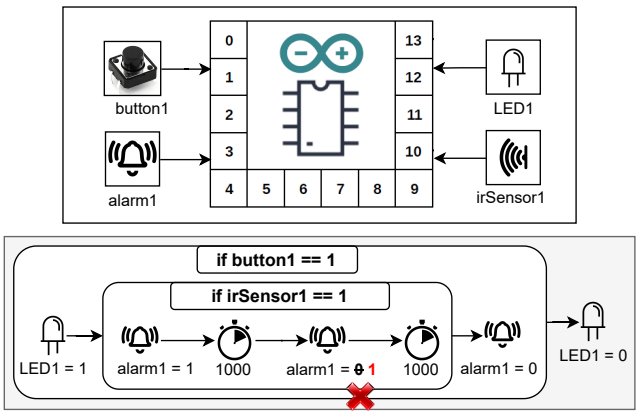

Figure 2. An example Arduino xModel. It has a defect since the alarm is not ringing as expected when the sensor detects an obstacle (it is highlighted in red where `alarm1` is mistakenly set to 1)

how such a runtime state varies over time. As aforementioned, the DigitalPin has a level attribute which represents the state of its Module, such as whether the button is pressed or the LED is on. This feature is the runtime state of an Arduino xModel because changes of its value at runtime put the model in different states.

Figure 1(**b**) provides some of the execution rules for the Arduino xDSL. In general, for every class of the xDSL's abstract syntax that has a runtime behavior, at least one execution rule must be defined to implement such behavior. For example, the press rule implements the behavior of pressing a PushButton. In accordance with the relationships between the classes of the abstract syntax, the execution rules may call each other as well to complement the xModel execution.

2.2.3 Behavioral Interface. A behavioral interface specifies how to trigger the execution of an xModel and/or how to interact with it during its execution [30], and is implemented by the execution rules. For instance, Figure 1(**c**) presents a behavioral interface for the Arduino xDSL. It comprises a set of events, each containing a set of parameters. An accepted event indicates a kind of request that can be accepted by an xModel and an exposed event determines an observable reaction of a running xModel. Accordingly, when executing an Arduino xModel, it is possible to request for pressing a button (using `button_pressed` event) and for detecting an obstacle by a sensor (using `IRSensor_detected` event). Whenever the value of the `level` feature of a DigitalPin changes during execution, it will be exposed by a `Pin_level_changed` event. With these facilities in place, we can simulate, debug, and test the behavior of an Arduino xModel (such as the one in Figure 2) before deployment.

2.2.4 xModel Execution Trace. An xModel's execution is captured in a trace that specifies which execution rules

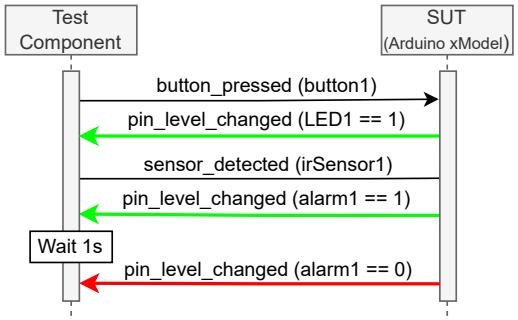

Figure 3. An example test case for the Arduino xModel

of its xDSL's semantics are called by which elements of the xModel [10, 11]. In Section 3.2.2, an example will be given.

2.3 Testing for xDSLs

xModels formally describe the behavior of a system. They can be executed in the xDSLs' supported modeling environment to simulate such behavior. When the environments offer V&V techniques (e.g., testing), it is also possible to ensure the correctness of the xModel's behavior as early as possible. For example, Khorram et al. proposed a testing framework for xDSLs [27]. Using this approach, we wrote a test case for the Arduino xModel of Figure 2. As displayed in Figure 3, it is defined as a scenario of exchanging data between a Test Component and the System-Under Test (SUT), in our case the Arduino xModel. The test data are instances of the events specified by the behavioral interface of the Arduino xDSL. The values of the events' parameters are elements of the Arduino xModel with values for their runtime features (such as `alarm1` element with value 1 for its `level` feature).

This test case checks whether the LED turns on when the button is pressed and whether the alarm alternates between noise/silence periods when the sensor detects an obstacle. Due to the defect of the Arduino xModel, the first two assertions pass but the last one fails; their corresponding arrows in Figure 3 are highlighted in green and red, respectively.

2.4 Coverage

Coverage is a popular test quality measurement technique that analyzes how much of the system under test is exercised by a given test case based on a given criterion. There are many coverage metrics in the literature, each observing the system execution from a different perspective. For example, in the context of programming languages, *Statement coverage* metric computes the percentage of the statements of the software that are executed and *Method coverage* metric calculates the percentage of the methods that at least one of their inner statements is executed [1].

2.5 Spectrum-Based Fault Localization (SBFL)

In the context of software debugging, SBFL is a popular fault localization technique that uses the results of test cases and their corresponding code coverage information to estimate the likelihood of each program component being faulty (using specific arithmetic formulas) [18]. Depending on how the coverage is computed (i.e., coverage metric), the examined components could be different. For instance, when SBFL uses statement coverage for a Java program, it calculates the probability of each program's statement to have a fault [49].

2.6 Motivation and Objectives

In the context of xDSLs, there are generic testing approaches that enable testing behavioral models [14, 26, 27, 34, 50]. To perform testing efficiently, there is a need to evaluate the quality of the defined test cases and to localize the model's faults in case of test failure. While these concerns are well-addressed in the field of software testing by coverage computation, it is still an understudied concern for xDSLs. Indeed, coverage computation is both a means for test quality measurement [1] and a foundation of SBFL techniques [49]. Leveraging coverage in the context of xDSLs is still a manual task and is hitherto performed for specific DSLs [37]. As there are plenty of xDSLs for designing systems of specific domains [5, 39, 41] and as engineering of additional ones for emerging domains is recurrent [33], there is a strong incentive to conceive a *generic* coverage metric that could be used to provide coverage computation and fault localization means for any xDSL.

Regardless of the xDSL used for the definition of an xModel, every xModel can be formally defined as a specific kind of graph in which model elements are defined as nodes, and different types of edges exist to specify containment, inheritance, and cross-references [6]. When executing an xModel, the result can systematically be captured in an execution trace, using a fixed and generic format, which keeps track of the xModel's exercised elements. Based on this perspective, and through an analysis of the xDSL definition itself, it is apparent that we can adapt the *node coverage* metric—from structural graph coverage criteria [1]—for the context of xDSLs, hence reasoning about the xModel coverage in a generic way. Indeed, we can define a new coverage metric that generally considers xModels' elements as software components to be covered. Consequently, we can then leverage SBFL for localizing the faulty elements of xModels based on such a coverage metric. Therefore, to offer a generic coverage computation and fault localization framework for xDSLs, we aim for the following objective:

Objective 1 Generic coverage metric which computes the coverage of xModels' test suites considering the xModels' elements as components to be covered.

In addition to the generic coverage metric, specific coverage aspects may be required for particular xDSLs. Thus, we also aim for the following objective:

Objective 2 Customization techniques for the generic coverage metric which allow dealing with the peculiarities of a specific xDSL while providing automation.

With coverage measurements at hand, we can offer fault localization for xDSLs by adapting existing SBFL techniques. Accordingly, we defined our third objective as:

Objective 3 Environment that supports the localizing of faulty model elements by computing the suspiciousness-based ranking of the models' elements using the coverage measurements.

3 Approach

In this section, we present our proposed framework.

3.1 Overview

Figure 4 displays an overview of our proposed framework. Two roles are involved: a language engineer (at the top left corner) who defines an xDSL according to the definitions given in Section 2.2, and a language user (at the top right corner) who defines xModels (using the xDSL) and test cases for them. We assume there is an existing testing framework (at the center) that (1) provides facilities for writing and executing test cases for xModels; and (2) produces the results of the tests along with the execution trace for the tested xModel (such as [27]).

The first component of our framework, namely the *xModel Coverage Computation* component, generates the coverage matrix of each xModel's test case (at the bottom left). The coverage matrix is a list of elements from the xModel along with their coverage status. Two sources of information are used by this component. First, from the definition of the given xDSL, the *xModel Coverage Computation* component recognizes which classes of the abstract syntax are used by each execution rule of the operational semantics. This is required to recognize what are the "traceable" elements of the xModel, i.e., elements whose execution may be captured in the trace. Second, the *xModel Coverage Computation* component analyzes the execution trace of the tested xModel to extract the xModel's elements that are captured in the trace, meaning that they are *covered* by the test case.

The framework allows the language engineers to optionally define specific coverage rules for their xDSL (at the bottom left corner). Accordingly, the *xModel Coverage Computation* component uses such rules, if they are available, to update the generated coverage matrix according to the specific needs of the xDSL.

Based on the computed coverage measurements, additional techniques may be applied, such as test suite minimization or fault localization. In this paper, we apply a set of collected SBFL techniques taken from [45]. Thus, we provide a second component (at the bottom center) that reads the test results produced by the test execution engine and the coverage matrix constructed by our *xModel Coverage Computation* component to generate the suspiciousness-based ranking of the xModel's elements. Such ranking helps in debugging the xModel as it directly positions the location of the faults.

Lastly, the framework provides a *Visualization* component, in which the test results, the computed coverage, and the suspiciousness ranking are visualized.

3.2 Coverage Computation

In this section, we introduce a new coverage computation metric for the context of xDSLs. It can be used for computing the coverage measures for any xModel, regardless of the xDSL used for its definition. To construct the coverage matrix for an xModel using this metric, we need information about the xModel execution. Such information can be accessed from two main sources, including the definition of the xDSL and the xModel execution trace.

3.2.1 Analyzing the xDSL Definition.

As explained in Section 2.2, given an xDSL, the execution of a conforming xModel is performed by calls to the execution rules of the xDSL's operational semantics. Each execution rule uses specific classes of the xDSL's abstract syntax. This means that, when running an xModel, the execution of its individual elements will be captured in a trace only if there is at least one execution rule defined for either a direct or inherited type of the element. Therefore, by analyzing the definition of an xDSL, we can identify the classes of its abstract syntax for which instances can be considered traceable, and thus, whose coverage by a test case can be detected using an execution trace. Algorithm 1 shows this analysis with an xDSL as input and a list of classes namely traceableTypes as output. Its output will be used for the coverage computation of the xModels which are defined by its input xDSL.

Algorithm 1: Finding the traceable types of an xDSL

```
   Input:
   xDSL.syntax: the abstract syntax of the xDSL,
   xDSL.semantics: the operational semantics of the xDSL
   Output :
   traceableTypes: classes of the xDSL's abstract syntax for which the
   execution of their objects can be traced
1  begin
2      foreach rule ∈ xDSL.semantics do
3          traceableTypes.add (rule.class)
       // Checking inheritance relationships
4      foreach class ∈ xDSL.syntax do
5          if class ∉ traceableTypes
6          ⋀ class.allSuperClasses → exists (c|c ∈ traceableTypes)
             then
7              traceableTypes.add (class)
```

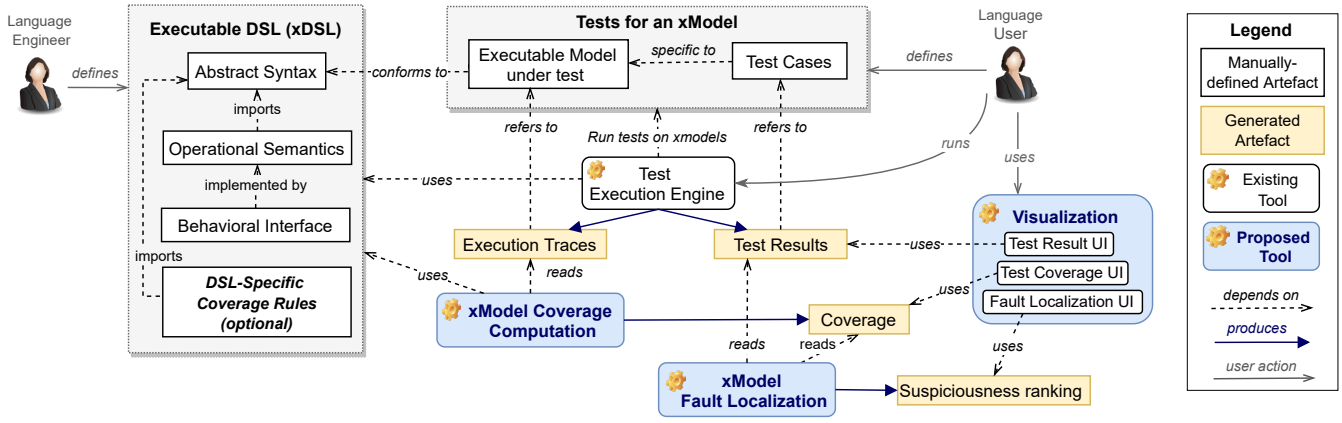

Figure 4. Approach Overview

3.2.2 Initializing the Coverage Matrix for the xModels' Tests.

After running a test case on an xModel, we compute its initial coverage using the xModel's execution trace. We described in Section 2.2 that such a trace is a sequence of called execution rules on the elements of the xModel [10, 11]. Therefore, by analyzing the trace, we can extract the xModel's elements covered by the test case.

For example, we can run the test case of Figure 3 on the Arduino xModel of Figure 2. This results in running the Arduino xModel itself which is performed by calling rules of the Arduino semantics (part (**b**) of Figure 1) as follows. When the test case sends a request for pressing `button1`, first the press(button1) rule is called, which results in a set of consecutive calls: execute(sketch), execute(if), and execute(LED1=1) that turns on the LED1 (calls changeLevel(LED1)) because the `button1` is pressed. Thus the first assertion passes (the first green arrow in Figure 3). Subsequently, the test case requests to put the `irSensor1` in the state of detecting an obstacle (sending sensor_detected(irSensor1) to the Arduino xModel). This results in a call of detect(irSensor1) which triggers the sequence execute(if), execute(alarm1=1) that turns on the alarm (calls changeLevel(alarm1)), execute(delay) that waits for 1000 milliseconds, execute(alarm1=1) that must turn off the alarm—but due to the defect it does not— and execute(delay) to wait 1000 milliseconds. So the second assertion passes but the third one fails (the second green arrow and the red arrow in Figure 3, respectively).

This set of calls is captured in an execution trace of the Arduino xModel as shown on the top of Figure 5. Using this trace, we can construct the initial coverage matrix of the test case of Figure 3. As displayed on the bottom of Figure 5, we consider the elements captured by the trace as "covered" (highlighted in green) and the rest (highlighted in yellow) will be examined in the next steps of coverage computation as described next.

3.2.3 DSL-Specific Coverage Rules.

As already mentioned in Section 2.6, some xDSLs may require specific coverage

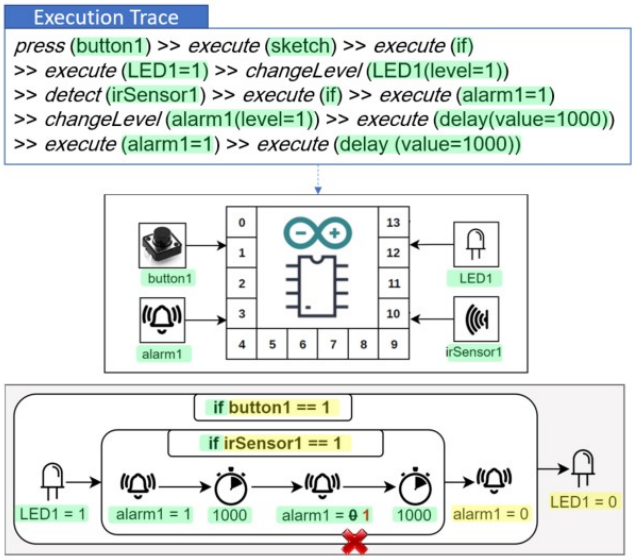

Figure 5. The coverage of Arduino xModel of Figure 2 by the test case of Figure 3 based on its execution trace (covered elements are highlighted in green, and yellow-highlighted elements will be examined in the next steps)

metrics for their particular domain [37]. For example, deciding if an element is covered is based on what we were able to observe in the execution, i.e., captured in a trace. However, it may also allow deducing that other elements (e.g., referenced, contained by elements in the trace) can be considered as covered as well.

To provide this customizability, our framework optionally allows a language engineer to define a set of *DSL-specific coverage rules* for a given xDSL (shown at the bottom left corner of Figure 4). More specifically, we propose a dedicated metalanguage for defining such rules whose concepts are presented in Figure 6. Given the abstract syntax of an xDSL in the form of a metamodel, a DomainSpecificCoverage can be

defined for different Contexts each pointing to a metaclass of the xDSL's abstract syntax. For each Context, several Rules can be defined and we are currently considering two families of rules:

- **Inclusion rules**: a covered object, may induce that other objects are covered as well (see CoverageOfReferenced and CoverageByContent rule types).
- **Exclusion rules**: an object is ignored from coverage computation under a certain condition (see Ignore and ConditionalIgnore rule types).

Given an object conforming to a Context (directly or by inheritance), each type of rule acts as follows:

CoverageOfReferenced. From the coverage of the given object, we infer the coverage of its referenced objects (i.e., the value of its reference feature). Accordingly, the type of the reference will be added to the list of `traceableTypes` (i.e., output of Algorithm 1).

CoverageByContent. Inferring the coverage of the given object from the coverage of its contained objects (i.e., the value of its containmentReference feature). The object is covered if:

- `multiplicity = ALL`: all of its contained objects are covered.
- `multiplicity = ONE`: at least one of its contained objects is covered.

This rule also updates the list of `traceableTypes` by adding the metaclass of the Context.

Ignore. The object will be ignored from coverage computation, by considering it as "not-traced", except when the rule specifies not to ignore it if it conforms to the subclasses of the context (ignoreSubtypes= false).

ConditionalIgnore. The object will be ignored from coverage computation, by considering it as "not-traced", when it is contained by an object that:

- `condition = INCLUSION`: conforms to one of the containerType classes.
- `condition = EXCLUSION`: does not conform to any of the containerType classes.

These rules are applied in order repeatedly until a fixed point is reached i.e., until the coverage matrix becomes steady.

For example, Listing 1 shows some of the rules we have defined for the Arduino xDSL case. The CoverageByContent rule specifies that a Block object is covered if at least ONE of its contained Instruction elements is covered. According to the definition of the Arduino xDSL semantics (Figure 1(**b**)), there is no execution rule for the Expression class and its conforming objects are evaluated inside the execute(If) and the execute(While) rules. According to this information, we defined a CoverageOfReferenced rule specifying that whenever an If object is covered, its condition that is an Expression

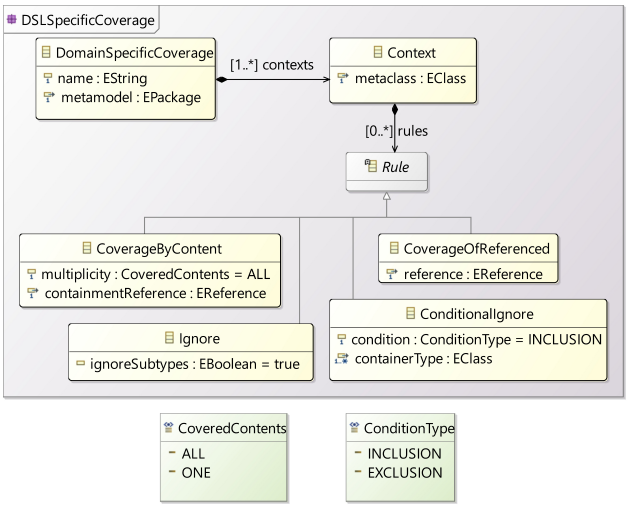

Figure 6. DSL-Specific Coverage Metamodel

```
1  DomainSpecificCoverage ArduinoCoverageRules{
2      Import metamodel arduino
3      Context Block{
4          CoverageByContent{
5              containmentReference instructions
6              multiplicity ONE
7          }
8      },
9      Context If{
10         CoverageOfReferenced {reference condition}
11     },
12     Context Module{
13         Ignore {ignoreSubtypes true}
14     }
15 }
```

Listing 1. Examples of Arduino-Specific Coverage Rules

is also covered. We also defined an Ignore rule to ignore instances of Module in the coverage computation as they are just representatives of physical elements.

3.2.4 Finalizing the Coverage Matrix for the xModels' Tests.

At the last step of coverage computation, we identify "not-covered" objects as follows. Given an object with an unspecified coverage status, it is "not-covered" if its type is traceable, i.e., contained in the traceableTypes list, and "not-traced" otherwise. Please note that we computed the traceableTypes in the previous steps, by analyzing the xDSL operational semantics (Section 3.2.1) and running the xDSL-specific coverage rules if we have any (Section 3.2.3).

Object	Type	Step1 Initial Coverage	Step2 Updated Coverage	Step3 Final Coverage
button1	Push-Button	**Covered**	**Not-Traced**	Not-Traced
if	If	**Covered**	Covered	Covered
button==1	Expression	-	**Covered**	Covered
alarm1=0	Module-Assignment	-	-	**Not-Covered**

Table 1. An excerpt of the coverage computation for the running example (changes of the step in bold)

Finally, we generate a complete coverage matrix for the whole test suite of the xModel by merging the coverage matrices produced for each of its test cases.

3.2.5 Generating a Coverage Matrix for the Running Example.

An excerpt of the result produced by each of the above-mentioned steps for some of the objects of the Arduino xModel of Figure 2 is provided in Table 1. As can be seen, (1) the `button1` object is considered as covered after trace analysis (Step 1), but is then ignored after updating the coverage matrix by the Arduino-specific coverage rules (Step 2); (2) the `if` object is covered based on the trace analysis (Step 1); (3) the `button==1` Expression does not have any status at first (Step 1) but it is then updated to covered after running the Arduino-specific coverage rules (Step 2) because when the `if` object is covered, its referenced expression element must be considered as covered; and, (4) the `alarm1=0` ModuleAssignment is not-covered by the test case (Step 3). At the end, the final coverage matrix is equivalent to the content of Table 1 modulo columns 3 and 4.

3.3 xModel Fault Localization

When a test case fails, it is hard to localize the defect causing the failure. Accordingly, various fault localization techniques are already proposed, like SBFL which is a coverage-based approach [49]. As our proposed coverage computation framework is generic regarding its supported xDSLs, it enables us to offer SBFL for any xDSL as well.

In the realm of software testing, SBFL is usually applied at the statement level, meaning that it uses the statement coverage of the program and calculates the suspiciousness of each statement [49]. In this paper, we adapt SBFL for the context of xDSLs, by substituting the notion of statement with the more generic concept of *element* of an xModel. Accordingly, considering a test suite of an xModel, our proposed *xModel Fault Localization* component uses the execution result and the coverage matrix of the test suite (i.e., generated by our *xModel Coverage Computation* component) to calculate the suspiciousness-based ranking of the xModel's elements using SBFL techniques. Generally, each SBFL technique introduces a formula that is based on a set of values (note that we adapted them for the context of xModels) which are computed from the test results and coverage information. For example, a well-known formula is Tarantula [25] defined as: $(NCF/NF)/(NCF/NF + NCS/NS)$ where:

- NCF: number of failed test cases that cover the element
- NCS: number of successful test cases that cover the element
- NS: total number of successful test cases
- NF: total number of failed test cases

SBFL follows the idea that the elements executed by more failed test cases are more likely to be faulty, and the ones executed by more passed test cases are less likely to have a fault. Our approach supports 18 existing formulas which have been collected by Troya et al. [45] by investigating primary studies proposing concrete SBFL techniques.

3.4 Definition of Artifacts

To preserve the genericity of our proposed framework, this section introduces a generic definition for its main artifacts including, the test result, the execution trace, and the coverage matrix for the xModels' tests. As Figure 7 shows, a test execution result is captured as a TestSuiteResult for each TestSuite, comprising a set of TestCaseResult for each TestCase with the value as PASS or FAIL. Each TestCaseResult provides a reference to the execution Trace of its tested xModel

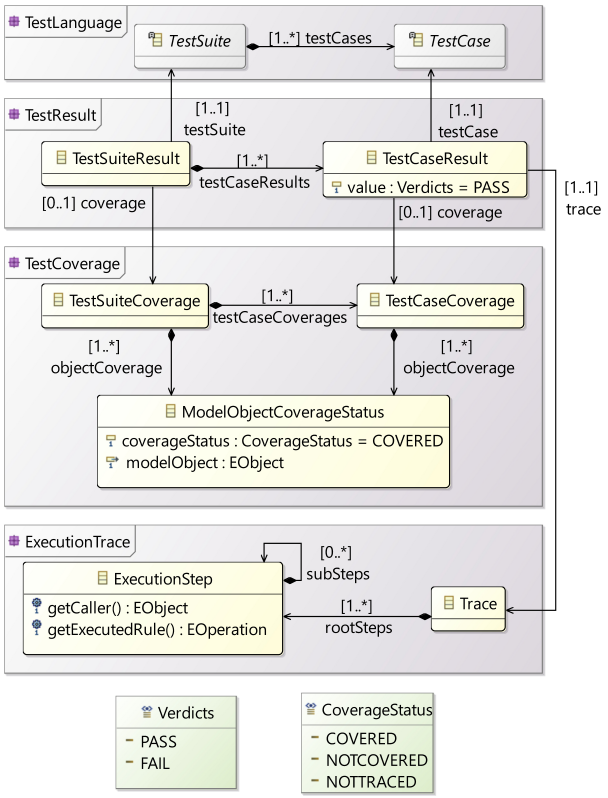

Figure 7. Definition of Artifacts

i.e., a set of ExecutionStep, each specifying what execution rule of the xDSL's semantics is called by which object of the tested xModel. Once coverage is computed for an executed test suite, a TestSuiteCoverage is generated which includes one TestCaseCoverage for each of its executed test cases. Both of them have a list of ModelObjectCoverageStatus instances, each specifies the coverage status of one object of the xModel for the test case/test suite. The coverage status is either COVERED, NOTCOVERED, or NOTTRACED.

4 Tool Support

We implemented our proposed framework as part of the GEMOC Studio [9]. It is a language and modeling workbench for xDSLs, which is defined on top of the Eclipse Modeling Framework (EMF) [44]. We used the testing framework proposed by Khorram et al. [27] which itself uses a generic execution trace management for xDSLs proposed by Bousse et al. [10, 11]. All the components of the framework (the *xModel Coverage Computation*, the *xModel Fault Localization*, and the *Visualization* components in Figure 4), are implemented in Java and are connected using the Eclipse extension point mechanism.

The suspiciousness computation implementation is based on that of provided by Troya et al. [45] for fault localization in model transformations, now adapted for general model elements. Currently, our tool supports 18 SBFL techniques but adding new ones is possible in our framework. Indeed within the literature, there are approximately 30 SBFL techniques [18, 38, 49]. They all use the set of values explained in Section 3.3 (i.e., NCF, NCS, NS, NF) to compute the suspiciousness-based ranking. Accordingly, any existing formula defined using the aforementioned variables can be added to the framework.

Figure 8 shows a screenshot of our tool running in the GEMOC Studio modeling workbench, after executing a test suite against the running example. The source code is accessible from a Zenodo repository [28]. In the project explorer on the left, there are two projects, one containing the Arduino xModel (shown in Figure 2) and another containing a test suite written for it using the testing framework proposed in [27]. All the artifacts of the tool can be persisted as XMI files conforming to the format presented in Figure 7 upon the request of the user—the user can select the related options in the run configuration. For example, Label 1 in Figure 8 indicates the generated files for the test execution result and the test coverage. For each executed test case, a copy of its model under test is also saved and its objects are referenced by the generated 'testCoverage.xmi' file. We provided a graphical view (label 2) for displaying the coverage measures computed for the test cases as well as for their test suite (at top center)[3]. For each element of the xModel under test, it shows its coverage status for all the tests, green for COVERED, red for NOTCOVERED, and yellow for NOTTRACED elements. Moreover, the last row (label 3) provides the percentage of the traceable elements covered by each test case and also by the whole test suite (i.e., 100 %). The user can also use two filter options, one to find all the elements with a specific coverage status (Coverage Filters on the left), and another to find the coverage status for a specific type of the elements (Model Element Filters on the left).

To run SBFL on the tested xModel, we provided another graphical view titled "fault localization" (label 4). It lists the traceable elements of the tested xModel, their coverage status by each test case, the test execution result (at the last row), and the required values for calculating the suspiciousness-based ranking. The view has a drop-down list of the 18 supported SBFL techniques (Label 5). When a technique is selected, the tool calculates the suspiciousness score and the rank for all the model elements and shows the results in the last two columns (label 6). Such ranking assists the language users in debugging their xModels by providing direct links to the location of the faulty elements.

For example, if we chose Phi as concrete SBFL technique, it calculates the first rank for the second ModuleAssignment of the second if condition of the Arduino xModel of Figure 2 where the defect is located (label 7). Therefore, the rank for the faulty element is correctly calculated. However, there are other elements with the same rank. This is a common output returned by SBFL techniques, due to the so-named *tied* elements [49]. There are some tie-breaking strategies in the literature which are left to future work to be studied for our context.

5 Evaluation

We performed an empirical study of our proposed framework to answer the following research questions:
RQ1: How much genericity is supported by the framework and how much customization is needed in order to have the intended coverage computations for xDSLs?
RQ2: To what extent is the result of the coverage computation component valid?
RQ3: Can the generically computed coverage measurements be used in fault localization techniques?

5.1 Experiment Setup

Setup for RQ1. For RQ1, we aim to investigate whether the framework can be used for different xDSLs. Accordingly, we chose four xDSLs from different domains:

- **xFSM**: A small language for designing Finite State Machines for processing strings.
- **xArduino**: A language for simulating Arduino boards and their execution logic (described in Section 2.1).
- **xPSSM**: A partial implementation of the Precise Semantics of UML State Machines (PSSM) [40] which supports modeling of discrete event-driven behavior.

[3]Note that the test results view is not shown here due to space limitations.

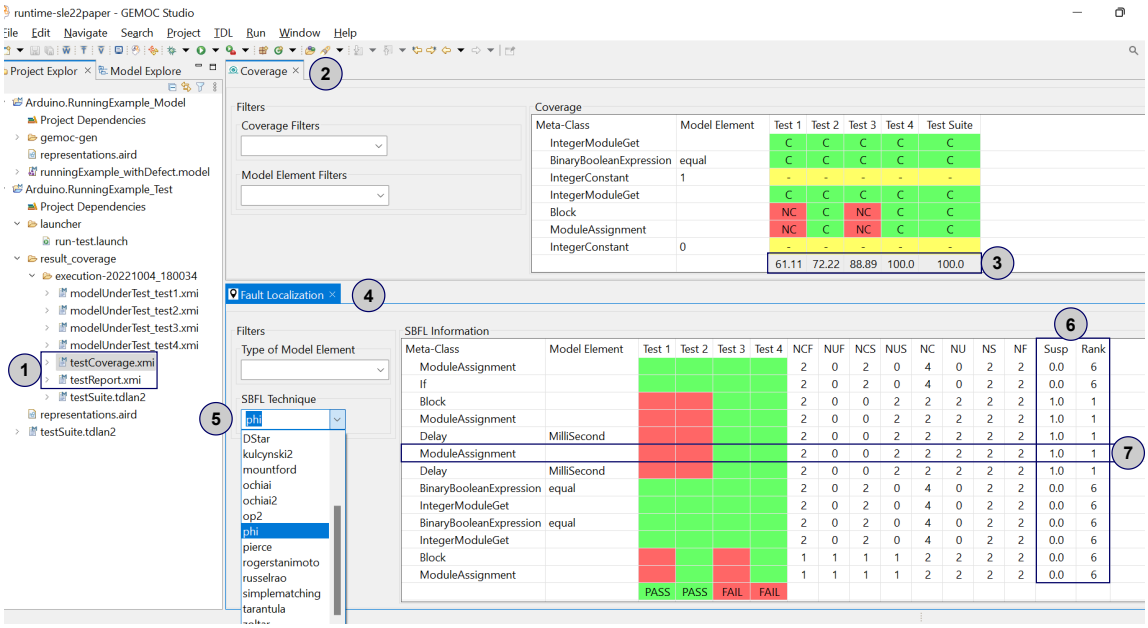

Figure 8. A screenshot of the provided tool running on the GEMOC Studio modeling workbench for the running example

- **xMiniJava**: A minimal implementation of Java based on the MiniJava project [12], allowing the definition of simple Java programs that can be executed directly by an execution engine rather than JVM. Note that it is not a typical xDSL and is defined just for experimental purposes as we will see in the following.

As presented in Table 2, the considered xDSLs have different sizes as the number of classes specified by their abstract syntax and the number of Lines of Code (LoC) of their semantics. On average, we defined 5 xModels using each xDSL in different sizes, ranging from 7 to 571 number of objects. In addition, using the testing framework of Khorram et al. [27], we wrote a set of test cases per xModel, altogether, 297 test cases for 21 xModels. The number of test cases for each xModel ranges from 3 to 81.

Answering the first research question also requires running the fault localization component on each considered xModel. To do this, there must be at least one defect in the xModel and at least one of its related test cases must be failing. A well-known technique for producing faulty programs is mutation in which small syntactic faults are injected into a program using so-called mutation operators. The result is a set of mutants that each is the program including some defects. In our experiment, we used WODEL [21], a generic mutation analysis framework which provides facilities to define mutation operators for an xDSL, then it automatically generates mutants for the xModels conforming to that xDSL.

As Table 2 presents, we defined cumulatively 184 mutation operators for our considered xDSLs, and WODEL generated a total of 1252 mutants for our 21 xModels. Afterward, we filtered the generated mutants by keeping those killed by their related test cases; a mutant is killed if at least one of its related test cases is failed on it. Among 1252 mutants, 1079 of them were killed by our written test suites.

Setup for RQ2. One way to answer RQ2 is to compare our coverage computation component with an existing coverage tool. As xMiniJava is a Java-like xDSL, each xMiniJava model is indeed a Java program and test cases of the xMiniJava models can be defined as JUnit tests for the equivalent Java programs. So we can compare our coverage computation approach with an existing Java coverage tool. For this comparison, we have chosen CodeCover as it is an open-source coverage tool supporting JUnit tests of Java programs [17]. Among different coverage metrics provided by CodeCover, we use *statement coverage* as it is the closest to our metric. CodeCover uses source code instrumentation approach to compute statement coverage.

We transformed test cases of xMiniJava models—according to Table 2, 77 tests for 6 xMiniJava models—to JUnit tests for equivalent Java programs. We reused the Java programs [13] provided by the MiniJava project.

Setup for RQ3. The third research question targets the usage of our coverage measurements for the fault localization component. For this, we need to assess whether our fault localization component correctly ranks the faulty element of an xModel as first. Accordingly, we used the 1079 killed mutants provided in the setup for RQ1, and to know the exact location of their injected fault, we used a tool named EMF Compare [16] to automatically find the faulty element of each mutant by comparing it with the original model. However, we are not aiming for an empirical evaluation of

		xFSM	xArduino	xPSSM	xMiniJava	Total
xDSL	Abstract syntax size (n. of EClasses)	3	59	39	76	-
	Operational semantics size (LoC)	111	768	975	1042	-
Tested xModels	Number of tested Models	5	6	4	6	21
	Size range of tested xModels (n. of EObjects)	7-133	18-59	61-154	31-571	7-571
Test Artifacts	Total number of test cases	45	22	153	77	297
	Test case numbers range of test suites	7-16	3-4	22-81	4-25	3-81
Mutation Analysis	Number of mutation operators	5	36	30	113	184
	Number of generated mutants	289	458	324	181	1252
	Number of killed mutants	194	457	308	120	1079
Coverage	Number of DSL-specific coverage rules	-	8	13	4	25
	DSL-specific coverage size (LoC)	-	50	75	26	151
	Number of test suites with computed coverage	45	22	153	77	297
	Range of computed test suites' coverage percentage	100%	100%	100%	98.08%-100%	-
SBFL	Number of fault localized mutants	194	452	304	119	1069

Table 2. Evaluation data at a glance

the performance of the different SBFL techniques and leave this to future work.

Evaluation data is available in our Zenodo repository [28].

5.2 Evaluation Result

Answering RQ1. In the first research question, we aim to evaluate whether the coverage computation and the fault localization facilities can be used for different xModels defined by various xDSLs. Specifically, we are questioning the level of required customization for each xDSL to have the intended coverage measurements for their conforming xModels. To answer RQ1, we used the prototype presented in Section 4 for 4 different xDSLs. For xArduino, xPSSM, and xMiniJava, their operational semantics do not provide enough information about the models' executions required for realizing the intended coverage measurements. To overcome this, we used the presented DSL for defining coverage rules, and in total, we have implemented 25 coverage rules of different types in 151 LoC. Therefore, using a few LoC, we efficiently realized the intended coverage computation for our xDSLs.

Next, we executed the test cases on xModels, a total of 297 test cases on 21 xModels, and we observed that their coverage has been computed successfully. To perform fault localization using the SBFL facility, we run the 297 test cases on the 1079 killed mutants, and then we used the SBFL techniques to get the suspiciousness ranking of the elements of 1069 mutants. As the examined xModels/mutants are defined using different xDSLs, it gives us the confidence to conclude that the framework provides the expected genericity feature.

Answering RQ2. This research question targets the validity of our proposed *xModel Coverage Computation* component. To answer it, we compared the coverage matrix generated by our proposed component for the MiniJava tests with that of generated by CodeCover for the statement coverage of equivalent JUnit tests. For example, Table 3 lists the coverage percentage for 5 randomly selected tests calculated by each tool. With our approach, it is calculated by dividing the number of covered model elements by the total number of traceable elements while with CodeCover is the percentage of covered Java statements. The slight differences between the results are because of some additional lines of code that CodeCover considers while they are not a statement (e.g., the closing curly brace of if statements). We manually verified that the coverage status of each Java statement by each JUnit test is the same for its equivalent MiniJava element by its related test, i.e., our approach provides the same result for the end user. This result shows the validity of our approach.

Answering RQ3. For this question, we aim to evaluate the usage of our generic coverage metric for subsequent tasks such as fault localization. For answering this question, we need to investigate whether our *xModel fault localization* component correctly finds the faulty element of each mutant using our computed coverage measures. Accordingly, we checked the result of running the SBFL techniques on 1079 killed mutants from RQ1 to see the rank of the mutants' faulty element calculated by each SBFL technique. We observed that for 1069 examined mutants (99 %), there is at least one SBFL technique that calculated the rank of its faulty element as first, hence emphasizing the usefulness of our

Test Cases	Our Coverage	CodeCover Coverage
test 1	23/33 = 69.70%	24/35 = 68.57%
test 2	7/36 = 19.44%	7/43 = 16.28%
test 3	28/49 = 57.14%	31/56 = 55.36%
test 4	51/54 = 94.44%	55/60 = 91.67%
test 5	46/119 = 38.66%	57/144 = 39.58%

Table 3. Coverage for a set of randomly selected tests

coverage measurement. As said earlier, we leave the performance evaluation of the different techniques as subject to future work, but we could show that in principle the framework allows employing SBFL techniques for xDSLs based on our coverage measurements.

5.3 Threats to Validity

We identify threats to validity according to the 4 main categories defined by Wohlin et al. [48] as follows.

Construct Validity. The validity of our coverage computation was compared with one existing coverage tool. There exist other tools like JaCoCo [24] and Cobertura [15]. In future work, we will compare with these tools to further support the validity of our code coverage computation.

Internal Validity. A recent survey on software fault localization [49] mentions that SBFL is incapable of locating bugs that are caused by missing code. Accordingly, we ignored mutation operators that define faults as removal of xModels' elements. This threatens the internal validity of our study. To overcome this threat, extensions of our framework with other fault localization techniques are required.

External Validity. In evaluating the genericity of our framework, we only considered four languages, so there is an external threat that the framework might not work as expected for other xDSLs. Additionally, we defined our framework considering the GEMOC Studio as a reference for xDSL implementations. As there are other language workbenches [20], additional studies are required to validate the portability of our approach.

Conclusion Validity. In answering RQ3, we observed that SBFL techniques can find the faulty element in an xModel. However, it is not clear which technique outperforms the other. This requires a deeper comparison between different SBFL techniques in an empirical evaluation to investigate their efficiency. This could also be useful in recommending the best techniques that offer the best ranking of the faulty elements given our coverage measurements.

6 Related Work

Several research efforts have proposed the use of existing coverage techniques for specific modeling languages, e.g., see logic coverage for State Machines [19], data-flow coverage for executable UML models [46], branch coverage for activity diagrams [7], among many others. To the best of our knowledge, there is no generic coverage criterion for xModels. Also, this topic is not yet discussed within the context of language workbenches [20].

Other proposals are related to the implementation of test coverage frameworks [8, 36, 43]. For example, Misurda et al. [36] propose a tool for testing Java programs based on execution paths to test coverage called *Jazz*. Bordin et al. [8] introduce their tool *Couverture* which is able to measure structural coverage by providing a virtualized execution platform. Sakamoto et al. [43] propose an extensible tool called *Open Code Coverage Framework* (OCCF), which supports a set of test coverage criteria. In addition, OCCF supports the addition of new test coverage as well as customization for new programming languages. OCCF has a similar goal as our approach, however, we rely on top of an existing tracing framework, which allows us to directly compute coverage measurements without instrumenting the xModels.

Some research efforts propose the application of SBFL to specific modeling languages [37]. For instance, some studies detect the faulty element in model transformations [31, 45]. Troya et al. [45] present an approach to apply SBFL to locate the faulty rule in a model transformation and evaluate the effectiveness of their approach by comparing a large set of different state-of-the-art SBFL techniques, which is also reused in the context of our work. Li et al. [31] propose an optimization strategy of SBFL by adding weight values to the test models as well as statistical coverage information. Raselimo & Fischer [42] present the usage of SBFL methods for context-free grammars based on a modified parser which collects grammar spectra, i.e., the covered rules for parsing a test case. We leave the application of our proposed framework for such domains subject to future work.

In addition to the approaches that target model transformations, some approaches target finding faulty elements in models. Wang et al. [47] propose the application of fault localization techniques for declarative models implemented in Alloy. Other approaches detect errors in models with the use of evolutionary algorithms. BLiMEA [4] and Ebro [3] detect errors in models based on evolutionary algorithms. Arcega et al. [2] compare these proposed tools for bug localization and show that the combination of these tools outperforms existing approaches. None of these approaches consider the operational semantics to detect errors, thus, our approach is complementary to the others, and vice versa.

To sum up, current approaches for coverage metrics and fault localization are mainly defined for GPLs or for one specific DSL. In this sense, with our framework, we aim to fill this gap by proposing a generic model element coverage metric for automatically computing coverage measures which may be used in subsequent steps such as fault localization.

7 Conclusions and Future Work

We proposed a generic coverage computation and fault localization framework for xDSLs. In our evaluation, we observed that an automated and customizable framework for coverage computation enriches the DSL definition with further V&V techniques at a reasonable cost.

As future work, we consider defining new coverage metrics, providing further support for the definition of DSL-specific coverage rules as well as detection of their conflicts, and investigating the efficiency of different SBFL techniques in the context of xDSLs.

Acknowledgments

This work has received funding from the European Union's Horizon 2020 research and innovation program under the Marie Skłodowska Curie grant agreement No 813884, by the Austrian Science Fund (P 30525-N31), and by the Austrian Federal Ministry for Digital and Economic Affairs and the National Foundation for Research, Technology and Development (CDG). Special gratitude to Javier Troya for putting his implementation of SFBL for model transformations as open source, and to Pablo Gómez-Abajo for his active support of the WODEL model mutation testing tool.

References

[1] Paul Ammann and Jeff Offutt. 2016. *Introduction to software testing*. Cambridge University Press.

[2] Lorena Arcega, Jaime Font Arcega, Øystein Haugen, and Carlos Cetina. 2021. Bug Localization in Model-Based Systems in the Wild. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 1 (2021), 1–32.

[3] Lorena Arcega, Jaime Font, and Carlos Cetina. 2018. Evolutionary algorithm for bug localization in the reconfigurations of models at runtime. In *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. 90–100.

[4] Lorena Arcega, Jaime Font, Øystein Haugen, and Carlos Cetina. 2019. An approach for bug localization in models using two levels: model and metamodel. *Software and Systems Modeling* 18, 6 (2019), 3551–3576.

[5] Reda Bendraou, Benoît Combemale, Xavier Crégut, and Marie-Pierre Gervais. 2007. Definition of an eXecutable SPEM 2.0. In *Proceedings of the 14th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE Computer Society, 390–397.

[6] Enrico Biermann, Claudia Ermel, and Gabriele Taentzer. 2008. Precise Semantics of EMF Model Transformations by Graph Transformation. In *Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems (MoDELS'08)*. Springer, 53–67. https://doi.org/10.1007/978-3-540-87875-9_4

[7] Pakinam N Boghdady, Nagwa L Badr, Mohamed A Hashim, and Mohamed F Tolba. 2011. An enhanced test case generation technique based on activity diagrams. In *Proceedings of the International Conference on Computer Engineering & Systems*. IEEE, 289–294.

[8] Matteo Bordin, Cyrille Comar, Tristan Gingold, Jérôme Guitton, Olivier Hainque, Thomas Quinot, Julien Delange, Jérôme Hugues, and Laurent Pautet. 2009. Couverture: an innovative open framework for coverage analysis of safety critical applications. *Ada User Journal* 30, 4 (2009), 248–255.

[9] Erwan Bousse, Thomas Degueule, Didier Vojtisek, Tanja Mayerhofer, Julien Deantoni, and Benoit Combemale. 2016. Execution framework of the GEMOC Studio (tool demo). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*. 84–89.

[10] Erwan Bousse, Tanja Mayerhofer, Benoit Combemale, and Benoit Baudry. 2015. A Generative Approach to Define Rich Domain-Specific Trace Metamodels. In *Modelling Foundations and Applications*, Gabriele Taentzer and Francis Bordeleau (Eds.). Springer, 45–61.

[11] Erwan Bousse, Tanja Mayerhofer, Benoit Combemale, and Benoit Baudry. 2019. Advanced and efficient execution trace management for executable domain-specific modeling languages. *Software and Systems Modeling* (2019), 1–37. https://doi.org/10.1007/s10270-017-0598-5

[12] Joao Cangussu, Jens Palsberg, and Vidyut Samanta. 2022. Modern Compiler Implementation in Java: the MiniJava Project. https://www.cambridge.org/resources/052182060X. [Online; accessed 30-September-2022].

[13] Joao Cangussu, Jens Palsberg, and Vidyut Samanta. 2022. Sample MiniJava Programs. https://www.cambridge.org/resources/052182060X/#programsl. [Online; accessed 30-September-2022].

[14] Pablo C. Cañizares, Pablo Gómez-Abajo, Alberto Núñez, Esther Guerra, and Juan de Lara. 2021. New ideas: automated engineering of metamorphic testing environments for domain-specific languages. In *SLE'21: 14th ACM SIGPLAN International Conference on Software Language Engineering*. ACM, 49–54.

[15] Cobertura. 2022. A code coverage utility for Java. http://cobertura.github.io/cobertura/

[16] EMF Compare. 2022. EMF Compare. https://www.eclipse.org/emf/compare. [Online; accessed 30-September-2022].

[17] Code Cover. 2022. Measurement under Java. http://codecover.org/documentation/references/javaMeasurement.html. [Online; accessed 12-October-2022].

[18] Higor A de Souza, Marcos L Chaim, and Fabio Kon. 2016. Spectrum-based software fault localization: A survey of techniques, advances, and challenges. *arXiv preprint arXiv:1607.04347* (2016).

[19] Mounia El qortobi, Amine Rahj, Jamal Bentahar, and Rachida Dssouli. 2020. Test Generation Tool for Modified Condition/Decision Coverage: Model Based Testing. In *Proceedings of the 13th International Conference on Intelligent Systems: Theories and Applications*. 1–6.

[20] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Laurence Tratt, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. 2015. Evaluating and comparing language workbenches: Existing results and benchmarks for the future. *Computer Languages, Systems & Structures* 44 (2015), 24–47. https://doi.org/10.1016/j.cl.2015.08.007

[21] Pablo Gómez-Abajo, Esther Guerra, Juan de Lara, and Mercedes G Merayo. 2020. Wodel-Test: a model-based framework for language-independent mutation testing. *Software and Systems Modeling* 20 (2020), 1–27.

[22] Pedro Rangel Henriques, Maria João Varanda Pereira, Marjan Mernik, Mitja Lenic, Jeff Gray, and Hui Wu. 2005. Automatic generation of language-based tools using the LISA system. *IEE Proc. Softw.* 152, 2 (2005), 54–69. https://doi.org/10.1049/ip-sen:20041317

[23] Junaid Iqbal, Adnan Ashraf, Dragos Truscan, and Ivan Porres. 2019a. Exhaustive Simulation and Test Generation Using fUML Activity Diagrams. In *Advanced Information Systems Engineering*, Paolo Giorgini and Barbara Weber (Eds.). Springer, 96–110.

[24] JaCoCo. 2022. JaCoCo Java Code Coverage Library. https://github.com/jacoco/jacoco

[25] James A. Jones and Mary Jean Harrold. 2005. Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE'05)*. ACM, 273–282. https://doi.org/10.1145/1101908.1101949

[26] Faezeh Khorram, Erwan Bousse, Jean-Marie Mottu, and Gerson Sunyé. 2021. Adapting TDL to Provide Testing Support for Executable DSLs. *Journal of Object Technology* 20, 3 (2021), 6:1–15.

[27] Faezeh Khorram, Erwan Bousse, Jean-Marie Mottu, and Gerson Sunyé. 2022. Advanced Testing and Debugging Support for Reactive Executable DSLs. *Software and Systems Modeling* (2022). https://hal.archives-ouvertes.fr/hal-03723920

[28] Faezeh Khorram and Antonio Garmendia. 2022. *Coverage Computation and Fault Localization for Executable DSLs Artefacts*. https://doi.org/10.5281/zenodo.7186664

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

[30] Dorian Leroy, Erwan Bousse, Manuel Wimmer, Tanja Mayerhofer, Benoit Combemale, and Wieland Schwinger. 2020. Behavioral interfaces for executable DSLs. *Software and Systems Modeling* 19, 4 (2020), 1015–1043.

[31] Pengfei Li, Mingyue Jiang, and Zuohua Ding. 2020. Fault localization with weighted test model in model transformations. *IEEE Access* 8 (2020), 14054–14064.

[32] Daniel Lübke and Tammo van Lessen. 2017. BPMN-Based Model-Driven Testing of Service-Based Processes. In *Enterprise, Business-Process and Information Systems Modeling*. Springer, 119–133.

[33] Tanja Mayerhofer and Benoit Combemale. 2018. The Tool Generation Challenge for Executable Domain-Specific Modeling Languages. In *Software Technologies: Applications and Foundations*, Martina Seidl and Steffen Zschaler (Eds.). Springer, 193–199.

[34] Bart Meyers, Joachim Denil, István Dávid, and Hans Vangheluwe. 2016. Automated testing support for reactive domain-specific modelling languages. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*. ACM, 181–194.

[35] Stefan Mijatov, Tanja Mayerhofer, Philip Langer, and Gerti Kappel. 2015. Testing Functional Requirements in UML Activity Diagrams. In *Tests and Proofs*, Jasmin Christian Blanchette and Nikolai Kosmatov (Eds.). Springer, 173–190.

[36] Jonathan Misurda, James A Clause, Juliya L Reed, Bruce R Childers, and Mary Lou Soffa. 2005. Demand-driven structural testing with dynamic instrumentation. In *Proceedings of the 27th International Conference on Software Engineering (ICSE)*. IEEE, 156–165.

[37] Muhammad Luqman Mohd-Shafie, Wan Mohd Nasir Wan Kadir, Horst Lichter, Muhammad Khatibsyarbini, and Mohd Adham Isa. 2021. Model-based test case generation and prioritization: a systematic literature review. *Software and Systems Modeling* (2021), 1–37. https://doi.org/10.1007/s10270-021-00924-8

[38] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. 2011. A model for spectra-based software diagnosis. *ACM Transactions on software engineering and methodology (TOSEM)* 20, 3 (2011), 1–32.

[39] OASIS. 2007. Web Services Business Process Execution Language Version 2.0. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html

[40] Object Management Group. 2019. *Precise Semantics of UML State Machines*. Retrieved October, 2022 from https://www.omg.org/spec/PSSM/1.0/

[41] Object Management Group (OMG). 2022. Semantics of a Foundational Subset for Executable UML Models. https://www.omg.org/spec/FUML/. (last accessed in September 2022).

[42] Moeketsi Raselimo and Bernd Fischer. 2019. Spectrum-based fault localization for context-free grammars. In *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering*. 15–28.

[43] Kazunori Sakamoto, Kiyofumi Shimojo, Ryohei Takasawa, Hironori Washizaki, and Yoshiaki Fukazawa. 2013. OCCF: A framework for developing test coverage measurement tools supporting multiple programming languages. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. IEEE, 422–430.

[44] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. 2008. *EMF: eclipse modeling framework*. Addison-Wesley.

[45] Javier Troya, Sergio Segura, José Antonio Parejo, and Antonio Ruiz-Cortés. 2018. Spectrum-based fault localization in model transformations. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 27, 3 (2018), 1–50.

[46] Tabinda Waheed, Muhammad Zohaib Z Iqbal, and Zafar I Malik. 2008. Data flow analysis of UML action semantics for executable models. In *European Conference on Model Driven Architecture-Foundations and Applications*. Springer, 79–93.

[47] Kaiyuan Wang, Allison Sullivan, Darko Marinov, and Sarfraz Khurshid. 2020. Fault localization for declarative models in Alloy. In *Proceedings of the 31st International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 391–402.

[48] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in software engineering*. Springer Science & Business Media.

[49] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A survey on software fault localization. *IEEE Transactions on Software Engineering* 42, 8 (2016), 707–740.

[50] Hui Wu, Jeff Gray, and Marjan Mernik. 2009. Unit Testing for Domain-Specific Languages. In *Domain-Specific Languages*, Walid Mohamed Taha (Ed.). Springer, 125–147.