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

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

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

Faezeh Khorram[a,*], Erwan Bousse[b], Antonio Garmendia[c], Jean-Marie Mottu[b], Gerson Sunyé[b], Manuel Wimmer[d]

[a]*CNRS, University of Rennes, Rennes, France*
[b]*IMT Atlantique, Nantes Université, École Centrale Nantes, France*
[c]*Departamento de Ingeniería Informática, Universidad Autónoma de Madrid, Spain*
[d]*CDL-MINT, Institute of Business Informatics - Software Engineering, Johannes Kepler University Linz, Austria*

Abstract

Test coverage is an effective technique to measure test case quality and to enable fault localization. However, for Executable Domain-Specific Languages (xDSLs), coverage metrics and associated tools are currently manually defined for each xDSL resulting in costly, error-prone, and non-reusable work.

To address this problem, we propose a novel *language-parametric test coverage framework for xDSLs*. We define two coverage metrics adapted to xDSLs: model element and branch coverage. For performing coverage measurements, we propose a generic technique which can be used out-of-the-box by domain experts using any xDSL to define, execute, and test models. In addition, the coverage of model elements and branches can be parameterized for a given xDSL through the definition of coverage rules using a dedicated language. We showcase two test coverage applications for xDSLs: measuring the quality of test suites for executable models and localizing the models' faults using Spectrum-Based Fault Localization techniques. We evaluate our approach using four different xDSLs. Results show that (i) we can generate meaningful coverage measurements for all investigated models, (ii) the provided coverage rule language enables framework parameterization for all xDSLs, and (iii) the computed coverage measurements are useful in identifying defects of the models.

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

1. Introduction

Domain-Specific Languages (DSLs) are commonly associated with a modeling workbench allowing the language users to define models. Using such workbenches, a common task is to describe the dynamic aspects of a system in terms of *behavioral models*. To confirm that these models fulfill their expectations, dynamic Verification and Validation (V&V) techniques are used. Such techniques require the possibility to *execute* a model, and thus, are only suitable for DSLs with execution semantics. We refer to such DSLs as *executable DSLs (xDSLs)*, which are the focus of this paper.

Testing, as a prevalent dynamic V&V technique, checks whether a system shows the expected behavior during its execution. Nowadays, testing frameworks for most General-purpose Programming Languages (GPLs) are available, and different testing approaches have already been proposed for xDSLs as well—some tailored for a specific xDSL [1, 2, 3, 4] and some providing generic solutions to be compatible with a wide range of xDSLs [5, 6, 7, 8, 9]. Yet, in addition to defining and executing test cases, two important concerns must be considered: (*i*) evaluating the quality of a given test suite and (*ii*) localizing the defects revealed by failed test cases. A popular solution for these concerns is *test coverage*, which aims at measuring which parts of a system were executed by a given test suite. It is both a metric for test quality evaluation [10] and an enabler for automatic fault localization, e.g., see Spectrum-Based Fault Localization (SBFL) [11].

To measure the coverage of test cases of programs defined by GPLs, different techniques exist, one of those, using the control-flow graph of a program along with program instrumentation for computing statement and branch coverage [10]. Unfortunately, as these techniques rely on concepts specific to the domain of GPLs (e.g., *statements*, *conditionals*, etc.), they cannot be directly transposed to DSLs which follow diverse computation paradigms (e.g., consider *states* and *transitions* of state machine based DSLs). Proposing new test coverage techniques for each new xDSL from scratch again and again is tedious and error-prone, hence a more reusable but at the same time adaptable solution supporting a wide range of xDSLs is required.

In this paper, we propose a novel *language-parametric test coverage framework for xDSLs*. The main contributions of this work include:

- Two coverage metrics for xDSLs: *model element coverage*

*Corresponding author
Email addresses: faezeh.khorram@inria.fr (Faezeh Khorram), erwan.bousse@ls2n.fr (Erwan Bousse), antonio.garmendia@uam.es (Antonio Garmendia), jean-marie.mottu@ls2n.fr (Jean-Marie Mottu), gerson.sunye@ls2n.fr (Gerson Sunyé), manuel.wimmer@jku.at (Manuel Wimmer)

and *model branch coverage*, measuring to which extent each model element and "branch" of the model was executed by a given test suite, respectively. Our proposed measurement technique constructs the coverage matrices of a test suite by analyzing the execution traces of its model under test.
- A dedicated Model Coverage Language (MoCL) that allows a language engineer to define a set of *coverage rules* for a given xDSL. These rules parameterize the framework for the xDSL by specifying what types of model elements and branches must be considered when computing the coverage of its conforming models.
- A showcase of two test coverage applications for xDSLs: measuring the quality of a test suite for an executable model, and localizing faults in an executable model by calculating the suspiciousness-based ranking of the model elements using existing SBFL formulas [12].

The proposed framework is implemented for the Eclipse GEMOC Studio [13], a language and modeling workbench for xDSLs. We conducted an empirical evaluation for four different xDSLs, totally 301 test cases for 21 executable models with sizes ranging from 7 to 571 elements and 1252 mutants generated by injecting faults into the models. We observed that for all the examined test suites, meaningful model element and model branch coverage measurements can be automatically constructed. The proposed language also enabled us to parameterize the framework for all considered xDSLs as well as to reproduce a set of existing code coverage tools. In addition, the model element coverage measurements allowed the application of existing SBFL techniques for tracking the faulty model elements (introduced by mutations). We also measured the effectiveness of the different SBFL techniques for the context of executable models by calculating the required effort for finding the faulty element of a model in the best-case, average-case, and worst-case scenarios. The results show that the different techniques provide varying performance for the considered xDSLs which makes the selection of the appropriate techniques out of the available ones for an xDSL necessary.

Please note that this paper is an extended and thoroughly improved version of our previous work [14]. New contents include proposing the *model branch coverage* metric along with its measurement technique, an extension of the proposed model coverage language to support conditional and branch coverage rules. This is a challenging problem because languages have different syntax and semantics which makes the analysis of branches complicated on a general level. To the best of our knowledge, there is no existing approach that proposes such a generic approach going beyond a specific language. Furthermore, we provide an extended evaluation to (*i*) assess the branch coverage computation and definition effort; (*ii*) measure the effectiveness of different SBFL techniques in the context of xDSLs; and (*iii*) compare our approach with two additional code coverage tools.

The rest of the paper is organized as follows. We provide the background and a running example in Section 2. Section 3 introduces an overview of the proposed framework which is then detailed in two follow-up sections (Sections 4 and 5). The supporting tool is presented in Section 6 and Section 7 describes the performed evaluation. Finally, related work is given in Section 8 and Section 9 presents the conclusions of the paper with an outlook on future work.

2. Background

In this section, we present the required background and a running example that will be used throughout the paper.

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++. However, an xDSL specifically defined for Arduino can help in developing the Arduino programs using dedicated concepts rather than technical C instructions. In this paper, we use a sample xDSL designed for modeling Arduino boards along with their behaviors as a running example. Next, we present the definition and usage of this Arduino xDSL based on the required language engineering artefacts.

2.2. Executable Domain-Specific Languages

A DSL with execution semantics is commonly referred to as an *executable DSL (xDSL)*. We call *executable model* a model that conforms to an xDSL. It is essentially a program written using an xDSL, and which can be executed according to the xDSL semantics. Please note that the term "model" from now on refers to an executable model. Additionally, we refer with the term *"language engineer"* to the person who is in charge of defining an xDSL, and finally, with the term *"domain expert"* to the person using the xDSL.

In the scope of this paper, we consider an xDSL has at least two parts: (*i*) an abstract syntax specifying the domain concepts along with the relationships between them; and (*ii*) an execution semantics enabling the execution of the models.

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 [15] which is similar to the core of UML Class Diagrams. The root element of the Arduino xDSL 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, PushButton, InfraRedSensor, and Buzzer. The DigitalPin has a level attribute which represents the state of its Module. For instance, 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

[1] `https://www.arduino.cc/`

[2] Inspired from `https://github.com/mbats/arduino`

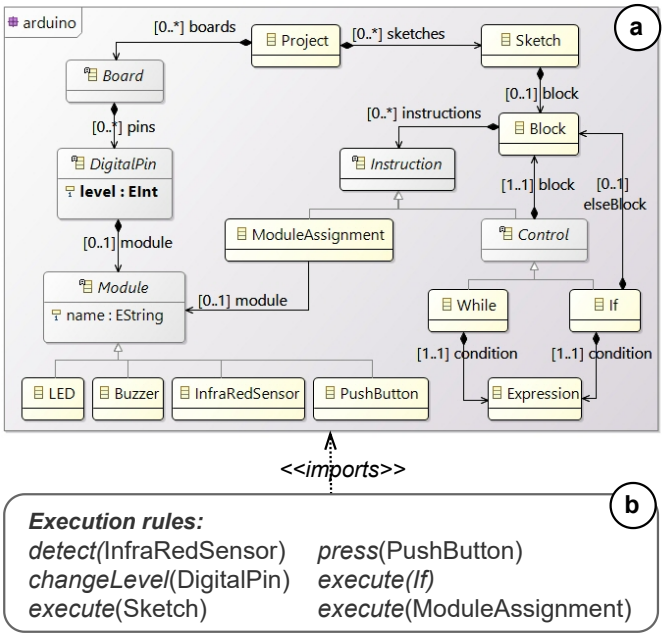

Figure 1: An excerpt of an xDSL defined for Arduino

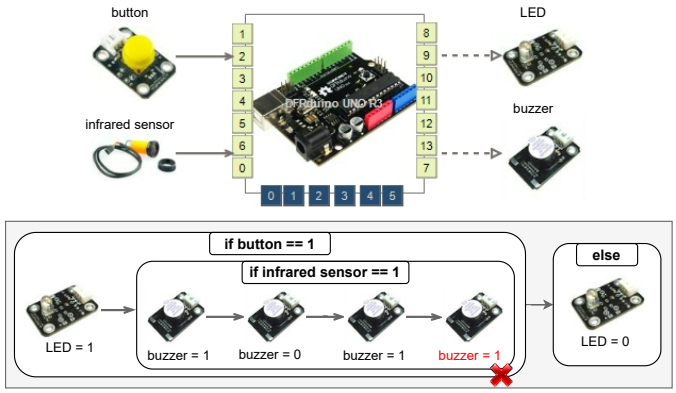

Figure 2: An example xArduino model representing a basic intrusion alarm system. It has a defect since the buzzer is not ringing as expected (it is highlighted in red where `buzzer` is mistakenly set to 1).

comprise several Instructions such as ModuleAssignment for changing the state of a Module, and Control instructions to define conditional behaviors (e.g.,, using If or While).

Figure 2 shows a sample Arduino model, defined by instantiating abstract syntax elements. At the top, there is an Arduino Board with 14 DigitalPins, four of them connected to Module instances: a PushButton, an InfraRedSensor, an LED, and a Buzzer. The behavior of the board is defined in a Sketch instance (shown in the bottom of Figure 2) as: "if `button` is pressed, the `LED` turns on (i.e., activating the alarm system), and then if `infrared sensor` detects an obstacle, the `buzzer` alternates between noise/silence two times (i.e., reporting an intrusion). Otherwise, the `LED` turns off". Note that (*i*) pressing a button and sensing an obstacle by a sensor means the level of their DigitalPins is equal to 1; and that (*ii*) turning on/off an LED and a buzzer are ModuleAssignment instances—'LED=1', 'buzzer=1', 'LED=0', and 'buzzer=0' notations in Figure 2. As the model contains several conditional elements (i.e., if and else elements), its execution can enter into different branches. We intentionally inject a defect in this model where `buzzer` should be set to 0 but it is mistakenly set to 1, meaning that the buzzer turns on but does not alternate between noise/silence states (highlighted in red in Figure 2).

2.2.2. *Execution Semantics*

The execution semantics of an xDSL defines the possible runtime states of a model under execution as well as a set of execution rules specifying 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 defines the runtime state of an Arduino model because its value may change during the execution.

Figure 1(**b**) lists 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 model execution.

The execution of a model can be captured in an execution trace that specifies which execution rules of the xDSL's semantics are called by which elements of the model [16, 17]. For example, if we execute the xArduino model of Figure 2 with the initial runtime state button.level = 1, sensor.level = 1, the resulting trace would be as follows: press(button), execute(sketch), execute(if), execute(LED= 1), changeLevel(LEDPin(level= 1)), detect(infrared sensor), execute(if), execute(buzzer= 1), changeLevel(buzzerPin(level=1)), execute(buzzer= 0), changeLevel(buzzerPin(level=0)), execute(buzzer= 1), changeLevel(buzzerPin(level=1)), execute (buzzer= 1).

2.2.3. *Language and Modeling Workbench*

A language workbench offers a set of facilities to develop new DSLs, and a modeling workbench allows using the DSL in practice [18]. The Eclipse GEMOC Studio is both a language and a modeling workbench for xDSLs, and we use it in this paper to specify and tool up xDSLs [13]. However, it must be noted that our considerations for xDSL development also apply to other xDSL engineering platforms. We would also like to stress that a seminal evaluation and comparison study [19] shows that the aspect of test coverage support has not yet been in focus of language workbenches.

2.3. *Testing for xDSLs*

When a model describes the dynamic aspects of a system (known as *behavioral* model), we need to ensure that it represents the correct behavior by applying V&V techniques such as testing. For example, we previously proposed a generic testing framework for xDSLs in [9]. Using this approach, we can write a test case for the Arduino model of Figure 2 similar to Figure 3. This test case is defined as a scenario of exchanging messages

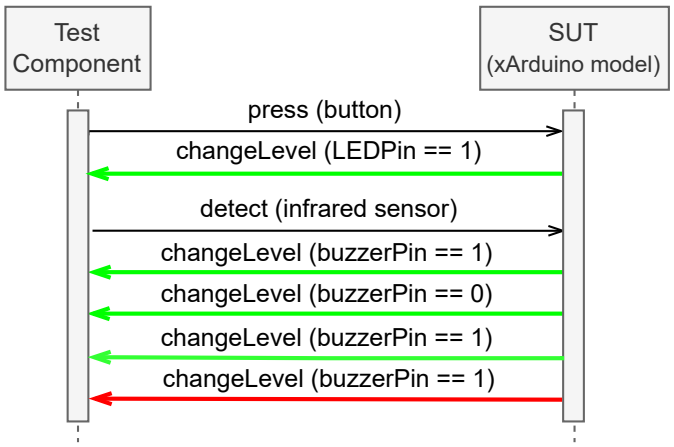

Figure 3: An example test case for the Arduino model of Figure 2

between the *test component*—i.e., the test executor—and the *System Under Test* (SUT)—in our case the Arduino model. When the sender is the test component, the exchanged message carries test input data, and the messages sent by SUT are assertions that carry the expected output. This test case checks whether the LED turns on when the button is pressed and whether the buzzer alternates between noise/silence periods when the sensor detects an obstacle. Due to the defect of the Arduino model, the first three assertions pass but the last one fails; their corresponding arrows in Figure 3 are highlighted in green and red, respectively.

2.4. Test Coverage

Coverage is a popular measurement technique that analyzes how much of the SUT is exercised by a given test case based on a given criterion. There are many coverage criteria in the literature, each observing the system execution from a different perspective. For example, in the context of GPLs, *statement coverage* metric computes the percentage of the program's statements that are executed, and *branch coverage* metric measures which possible branches of the control-flow structure of the program are followed [10]. Test coverage is highly used in the literature as both a metric for test quality evaluation [10] and an ingredient for automatic fault localization [11].

2.5. Spectrum-Based Fault Localization (SBFL)

SBFL is a popular automatic fault localization technique capable of locating different kinds of bugs, except those caused by missing code. It uses the results of test cases and their corresponding code coverage measurements to estimate the likelihood of each program component of being faulty by using specific arithmetic formulas [20]. Depending on the program's language and how the coverage is computed (i.e., the chosen coverage metric), the examined components can be different. For instance, when SBFL uses statement coverage for a given Java program, it calculates the probability of each statement of this program having a fault [11]. In Section 5.2, we provide more details on how SBFL works.

3. Objectives, Approach Overview, and Assumptions

In this section, we first talk about the motivation and the objectives of this paper (Section 3.1). Then, we present an overview of our proposed test coverage framework (Section 3.2) along with precise definition for the key concepts it relies on (Section 3.3). In the end, we further clarify the assumptions of the proposed approach (Section 3.4).

3.1. Motivation and Objectives

Measuring test coverage can lead to improving test efficiency as it provides both, a metric for test quality evaluation and enables fault localization in case of test failure. Despite the outstanding advancements in measuring test coverage for GPLs, providing test coverage for xDSLs is yet studied for only a few specific domains [21]. The main reason is the difficulty of adapting existing code coverage metrics for xDSLs, which originated from the dependency of their measurement techniques on the specific syntax and semantics of each language. For example, to compute branch coverage for a given program, we first need to identify the conditional constructs of its conforming language. Moreover, due to the huge amount of xDSLs for many existing [22, 23, 24] and emerging domains [25], there is a strong incentive to conceive a *generic* test coverage framework that could be used to provide test quality evaluation and fault localization means for any xDSL.

Regardless of the xDSL used for the definition of a model, every model can be formally defined as a specific kind of graph in which model elements are nodes, and different types of edges exist to specify relationships between elements such as containment, inheritance, and cross-references [26]. When executing a model, the result can systematically be captured in an execution trace, using a fixed and generic format, which keeps track of the model'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 [10]—for the context of xDSLs, hence reasoning about the model coverage in a generic way. We can thus define a coverage metric that generally considers model elements as components to be covered, to be called *model element coverage* metric. Therefore, to offer a generic test coverage framework for xDSLs, our first objective is as follows:

> **Objective 1:** A generic *model element coverage* metric to compute the coverage of elements of a model during the execution of a test suite.

In addition to *model element coverage*, further test coverage measures can be computed from other perspectives (i.e., based on other metrics). In the software testing area, *branch coverage* is one of the most popular metrics which is highly used in test case generation and amplification approaches [27]. A generic test coverage framework would allow a domain expert to compute branch coverage of its models under test without specifying what are branches in these models. In the same way, language

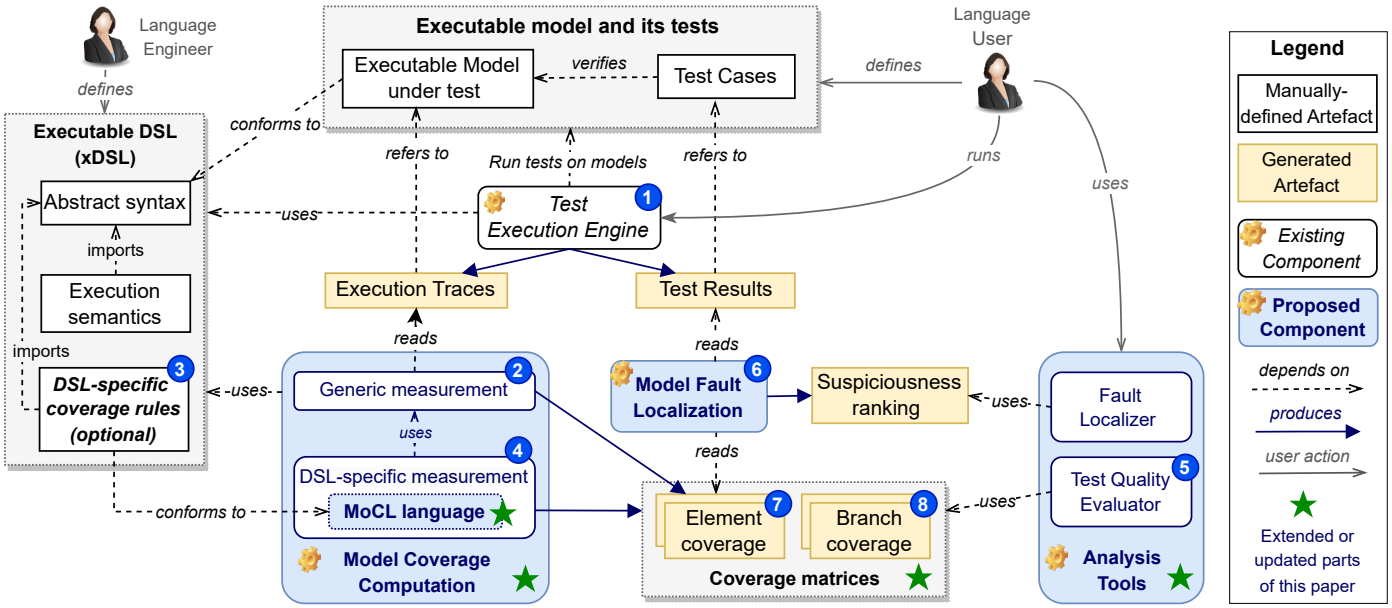

Figure 4: Approach overview (stars show which parts are improved as part of an extension of our previous work [14])

engineers expect help to only specify how to identify branches for their xDSLs, without requesting them to implement how to compute the *branch coverage*. Hence, while identifying the execution branches of the model totally depends on the definition of its conforming xDSL, test coverage framework may generically provide help to define and compute the branch coverage for any xDSL. Accordingly, we consider the following objective for our proposed framework:

> **Objective 2:** A generic *model branch coverage* metric to compute the coverage of branches of a model during the execution of a test suite.

Such preliminary computation requests the identification of what elements can be considered as "branches" given the semantics of the xDSL. Measuring test coverage based on the xDSL definition and the model execution traces may not always meet the specific coverage aspects of model elements of a particular xDSL. For instance, it is required to define what elements ought to be ignored from the coverage computation. Thus, we also aim for the following objective:

> **Objective 3:** Making the requested metrics *language-parameterizable* by extending and analyzing the xDSL specification to deal with the language specifics in computing the coverage of its conforming models.

With coverage measurements at hand, we can use them for test quality evaluation as well as fault localization of models using existing SBFL techniques. Hence, we defined our last objective as:

> **Objective 4:** An environment that (i) computes test quality based on the model element coverage and the model branch coverage metrics; and (ii) supports localizing faulty model elements by computing their suspiciousness-based ranking using the model element coverage measurements.

3.2. Approach Overview

Figure 4 displays an overview of our proposed framework. Two actors 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 domain expert (at the top right corner) who defines models (using the xDSL) and test cases for them. We assume there is an existing testing framework (label 1) that (*i*) provides facilities for writing and executing test cases for models; and (*ii*) produces the results of the test cases along with the resulting execution trace for the tested model (such as proposed in our previous work [9]).

Test coverage is computed in two phases realized by the *Model Coverage Computation* component of the framework. The first phase, called *Generic measurement*, computes test coverage based on the *model element coverage* criterion (label 2). It is an off-the-shelf approach that only requires two existing sources of information. First, from the definition of the given xDSL, it recognizes which classes of the abstract syntax are used by each execution rule of the execution semantics. This is required to recognize what are the "traceable" elements of the model, i.e., elements whose execution will be captured in the trace. Second, it analyzes the execution trace of the tested model to extract the model elements that are captured in the trace, meaning that they are *covered* by the test case. Accordingly, the "traceable" elements that are not captured in the trace are indeed *not-covered* by the test case. The final output is an *element coverage matrix* i.e., a list of model elements with their coverage status (label 7).

The second phase, called *DSL-specific measurement*, allows the language engineers to define DSL-specific coverage rules for their xDSLs (label 3) using the Model Coverage Language (MoCL). Two main types of rules can be defined for a given xDSL: customization rules to achieve an intended *model element coverage* measurement, and specification rules to adapt *branch coverage* metric for the xDSL. The proposed approach computes DSL-specific measurements (label 4) by applying the coverage rules on the generic coverage measurements and produces one coverage matrix per coverage rule-set: updating elements coverage (label 7) and returning branch coverage (label 8).

Lastly, we showcase possibilities offered by coverage measurements in two ways: (*i*) test quality evaluation by calculating the coverage percentage for each constructed coverage matrix and visualizing the results for the domain expert (label 5); and (*ii*) model fault localization based on SBFL techniques (label 6) by calculating the suspiciousness-based ranking of the model's elements using the test results produced by the test execution engine and the element coverage matrix constructed by our *Model Coverage Computation* component. This ranking may help to debug the model since possible failures are shown with an ordered ranking from highest to lowest probability.

3.3. Metamodel of Key Concepts

Among the concepts used in our framework, some must be precisely adapted for the context of this paper. As mentioned above and also illustrated in Figure 4 (label 1), any existing test execution engine that runs test cases on models and produces the model execution traces along with the test execution results can potentially be integrated into our test coverage framework. Both to clarify these requirements and to provide the structure of the elements manipulated or constructed by the approach (e.g., coverage matrices) we provide a precise definition of the key concepts using an Ecore metamodel shown in Figure 5.

Considering a TestLanguage which allows defining TestSuites each containing a set of TestCases, once the test suites are executed on a model, we expect the result to be captured as a TestSuiteResult and a TestCaseResult (with a 'PASS' or 'FAIL' verdict), for each TestSuite and TestCase, respectively. Each TestCaseResult should provide a reference to the execution Trace of its tested model. A Trace is a set of ExecutionStep, each specifying what execution rule of the xDSL's semantics is called by which object of the tested model.

If the test execution finishes without any problem meaning that the test engine provides the test results and the execution traces properly, our proposed *Model Coverage Computation* component measures the test coverage and constructs the coverage matrices as follows. For each coverage matrix constructed for an executed test suite, we generate a TestSuiteCoverage comprising one TestCaseCoverage per each executed test case. Both of them have a list of ModelObjectCoverageStatus instances, each specifying the coverage status of one model object for the test case/test suite. More specifically, for the *model element coverage* matrices, there is one ModelObjectCoverageStatus instance per model element. Similarly, for the *model branch coverage* matrices, there is one ModelObjectCoverageStatus for each branching root object. This object contains

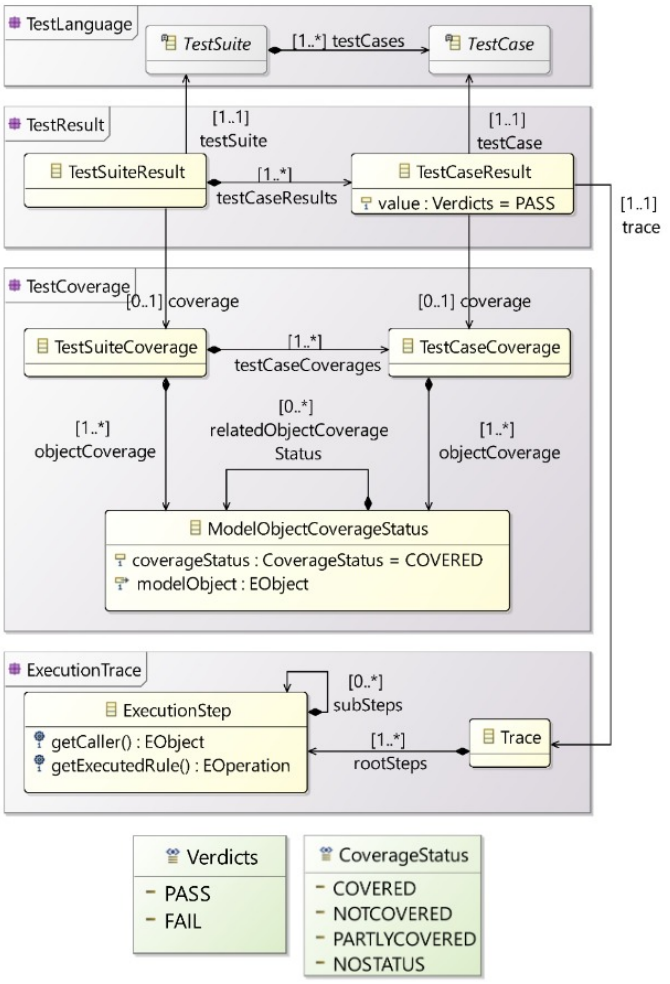

Figure 5: Metamodel of the key concepts required by the framework

several ModelObjectCoverageStatus instances. Each instance defines the coverage status of one of its originating branches The coverage status of an element is either 'COVERED', 'NOTCOVERED', 'PARTLYCOVERED', or 'NOSTATUS'. We will explain their meaning in the following sections.

3.4. Critical Reflection on Main Assumptions

In Section 2.2, we presented the used foundations for the definition of xDSLs for this paper. However, xDSLs can still differ in several other aspects which may affect the applicability of our proposed approach. For example, analogous to programming languages, there are both declarative and imperative xDSLs, or the execution semantics of an xDSL can be translational (i.e.,, compiled xDSL) or operational (i.e., interpreted xDSL). What matters for the applicability our proposed approach is if the given xDSL meets two main conditions. First, the execution traces of its conforming models are known and available. This means that for a model's execution, the xDSL must determine which model elements can be executed (i.e., are "traceable") among those which are indeed executed. The provided execution trace must be expressible in the format described in Section 3.3. For instance, to support a translational xDSL, the execution traces of the target language must be translated to the execution traces

of the source language, i.e.,, the xDSL. Second, there must be a test execution engine supporting (*i*) the execution of test cases on the xDSL's conforming models; and (*ii*) generating the test execution results as described in Section 3.3.

We consider the proposed test coverage framework as being *generic*, even if the definition of DSL-specific coverage rules is unsurprisingly language-specific. Indeed, the Model Coverage Computation component (Figure 4) is only defined once and is available for any xDSL. In addition, the *DSL-specific coverage rules* are optional and a default model element coverage is already provided out-of-the-box. Moreover, the framework provides a language to define the DSL-specific coverage rules, only describing how to consider the coverage of the elements modeled in the abstract syntax, without requesting the language engineer to implement any coverage rules.

4. Test Coverage Measurement

This section describes the two proposed coverage computation approaches, the generic approach in Section 4.1 and the DSL-specific approach in Section 4.2.

4.1. Generic Coverage Computation

In this section, we introduce an off-the-shelf coverage computation approach for the context of xDSLs. This approach computes coverage for any model regardless of the xDSL used for its definition. To do this, information is required about the model execution which can be accessed from two main sources: the xDSL definition and the model execution traces.

Algorithm 1: Finding the traceable types of an xDSL

```
   Input:
   xDSL.syntax: the abstract syntax of the xDSL,
   xDSL.semantics: the execution 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)
4      foreach class ∈ xDSL.syntax do
5          if class ∉ traceableTypes
6          ∧ class.allSuperClasses → exists (c | c ∈ traceableTypes)
             then
7              traceableTypes.add (class)
```

4.1.1. Analyzing the xDSL Definition

As explained in Section 2.2, a model conforming to a given xDSL is executed by calls to the execution rules of the xDSL's semantics. Each execution rule is defined for a specific class of the xDSL's abstract syntax. This means that the execution of those individual elements will be captured in the trace if there is at least one execution rule defined for either their direct or inherited type. 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 is used for computing the element coverage of the models which are defined by its input xDSL. For example, for the Arduino xDSL (Figure 1), the traceableTypes are: InfraRedSensor, PushButton, If, Sketch, and ModuleAssignment.

4.1.2. Computing Element Coverage Using Models Execution Traces

After running a test case against a model, we compute the element coverage of the test case using the traceable types (i.e., identified by Algorithm 1) and the model's execution trace. As shown in Algorithm 2, the model elements captured in the trace are covered by the test case (lines 2-4), the rest of the *traceable* elements are not-covered (lines 5-6), and for the not-traceable elements, no coverage status can be assigned (lines 7-8).

Algorithm 2: Computing model element coverage

```
   Input:
   traceableTypes: the output of Algorithm 1,
   model: the tested executable model,
   trace: list of model elements captured by the execution trace
   Output :
   elementsCoverages: a mapping of the model elements and their
   coverage status
1  begin
2      foreach element ∈ model.objects do
3          if element ∈ trace then
4              elementsCoverages.put(element, COVERED)
5          else if element.type ∈ traceableTypes then
6              elementsCoverages.put(element, NOTCOVERED)
7          else
8              elementsCoverages.put(element, NOSTATUS)
```

For example, if we run the test case of Figure 3 on the Arduino model of Figure 2, it results in running the Arduino model itself by calling the rules of the Arduino semantics (part (**b**) of Figure 1) as follows. When the test case sends a request for pressing `button`, first the press(button) rule is called, which results in a set of consecutive calls: execute(sketch), execute(if), and execute(LED=1) that turns on the `LED` by calling changeLevel(LEDPin (level=1)) because the `button` is pressed.

Next, the test case requests to put the `infrared sensor` in the state of detecting an obstacle. This results in a call of detect(infrared sensor) which triggers the sequence execute(if), execute(buzzer=1) that turns on the alarm (by calling the changeLevel(buzzerPin (level = 1))), execute(buzzer=0) that turns off the alarm (by calling the changeLevel(buzzerPin (level = 0))), execute(buzzer=1), that turns on the alarm for the second time, and execute(buzzer=1), that must turn off the alarm for the second time but due to the defect it does not.

This sequence of calls is captured in an execution trace as shown on the top of Figure 6. Using this trace, we can construct the element coverage matrix of the test case of Figure 3. As displayed on the bottom of Figure 6:

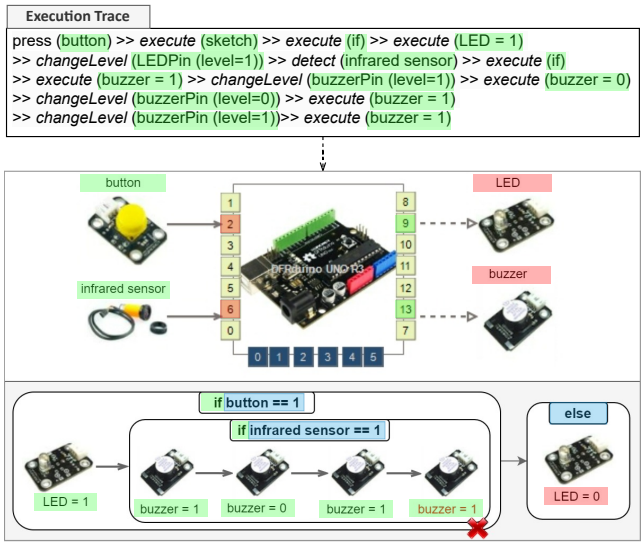

Figure 6: The element coverage of the Arduino model of Figure 2 by the test case of Figure 3 based on its execution trace (covered, not-covered, and not-traceable elements are highlighted in green, red, and blue, respectively.)

- the elements captured by the trace are "covered" (highlighted in green);
- the traceable elements that are not captured by the trace are "not-covered" (highlighted in red); and
- the not-traceable elements do not have any status (highlighted in blue).

At the end, we generate a complete element coverage matrix for the model's test suite by merging those produced for each of its test cases. The matrix can be persisted as a XMI file, conforming to the metamodel presented in Figure 5 (as presented later, Section 6).

4.1.3. Limitations of the Generic Coverage Computation

The coverage matrix produced by the generic approach may not be what the tester (i.e., the language user) expects. For example, the generated element coverage matrix for the running example (shown at the bottom of Figure 6), unexpectedly defines different coverage statuses for a DigitalPin and its connected Module (such as the `button`, covered and highlighted in green, whereas its `Digital Pin 2` is uncovered and highlighted in red). It also considers the Expression elements (e.g., button == 1) without any coverage status while it is reasonable to consider them as covered because their container Control element is covered (the `if`).

This unexpected result is due to the dependency of the approach on the definition of the xDSL's execution semantics. More specifically, depending on the classes of the abstract syntax for which execution rules are defined, and on how these rules call each other, the execution trace of a conforming model comprises different information, hence its element coverage matrix may be more precise. While changing the xDSL's definition could solve the issue, it may also affect other dependent tools such as model debuggers [28].

In addition, as already mentioned in Section 3.1, we can benefit from the power of *branch coverage* metric in the context of xDSLs if the language engineers specify how to identify branches for their xDSLs. Meaning that particular techniques are needed to specify how to adapt *branch coverage* for a given xDSL. In the following subsection, we propose a new approach to address both challenges without any need for changing the xDSL definition.

4.2. DSL-Specific Coverage Computation

The generic coverage computation approach decides if an element is covered based on what we were able to observe in the execution, i.e., captured in a trace. However, we may also be able to deduce that other elements, such as referenced or contained by the elements in the trace, are covered as well. In addition, if we can deduct what is an execution branch for a model based on its conforming xDSL, we can then leverage the branch coverage metric for the xDSL. To provide this customizability, our framework allows a language engineer to define a set of *DSL-specific coverage rules* for a given xDSL (as shown in Figure 4 (label 3)).

4.2.1. Defining DSL-Specific Coverage Rules

We propose a dedicated language for defining coverage rules for a given xDSL whose concepts are presented in Figure 7. 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 three families of rules:

- **Inclusion rules**: a covered element, may induce that other elements are covered as well (see CoverageOfReferenced and CoverageByContent rule types).
- **Exclusion rules**: an element is ignored from coverage computation under a certain condition (see Ignore and LimitedIgnore rule types).
- **Branch Specification rules**: specifying if an element originates different execution branches of the model. By this rule, the language engineer determines how to compute *branch coverage* for a model's test suite.

It is allowed to define conditional rules using Object Constraint Language (OCL) constraints by assigning a Condition element to the rules. Accordingly, given an element conforming to a Context (directly or by inheritance) that satisfies the rule's condition (if any), each type of the rule acts as follows:

CoverageOfReferenced Inclusion Rule. From the coverage of the given element, we infer the coverage of its referenced elements (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 Inclusion Rule. Inferring the coverage of the given element from the coverage of its contained elements (i.e., the value of its containmentReference feature). The element is covered if:

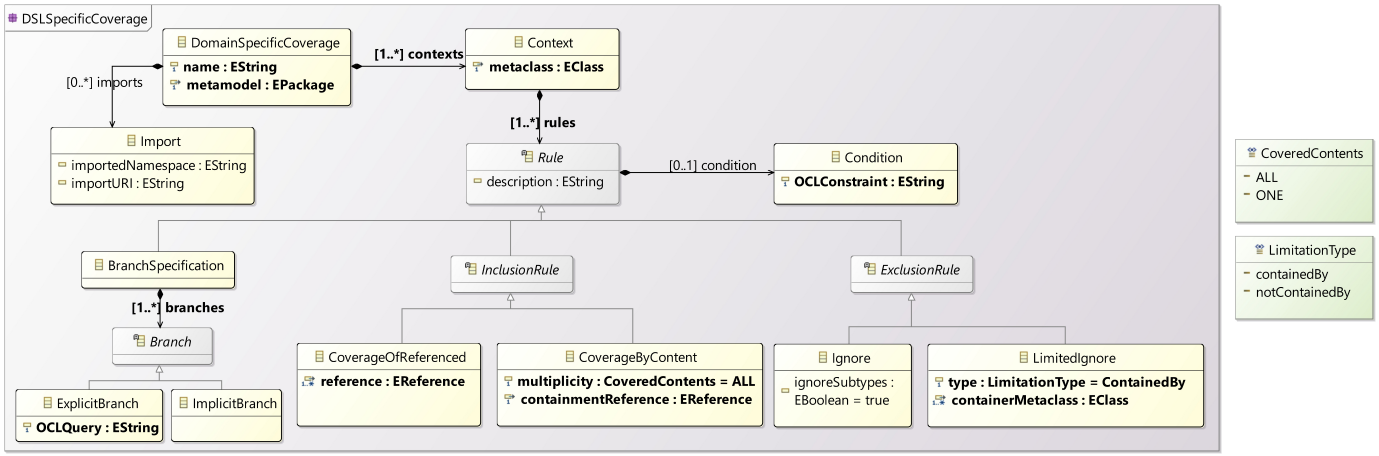

Figure 7: Abstract syntax of the Model Coverage Language (MoCL) to define DSL-Specific coverage rules

- `multiplicity= ALL` of its contained elements are covered.
- `multiplicity= ONE` of its contained elements is covered.
- `multiplicity= ANY` of its direct or indirect contained elements are covered.

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

Ignore Exclusion Rule. The element will be ignored from coverage computation, by changing its status to NO-STATUS, except when the rule specifies not to ignore the element if it conforms to the subclasses of the context (ignoreSubtypes= false).

LimitedIgnore Exclusion Rule. The element will be ignored from coverage computation, by changing its status to NO-STATUS if it is:

- `type= containedBy` an element that conforms to one of the containerMetaclasses.
- `type= notContainedBy` an element that conforms to any of the containerMetaclasses.

BranchSpecification Rule The element is a branching root element if it originates one to many execution branches. For instance, modeling a "while" with one branch, an "if..then..else" with two branches, a "Case" with several branches. A Branch can be of the following types:

- ExplicitBranch: A particular element of the model represents the execution branch and can be retrieved using an OCL query. Therefore, we can specify the coverage of the branch based on the coverage of its representative element.
- ImplicitBranch: There is an execution branch that cannot be specified by an element of the model (e.g., when an xDSL allows modeling "if..then" without "else"). In Section 4.2.2, we explain how to compute its coverage status.

For example, Listing 1 shows some of the rules we have defined for customizing the model element coverage for the Arduino xDSL. The Arduino metamodel is imported first (line 2) to get access to its metaclasses and their features. To infer

```
1  ruleset ArduinoModelElementCoverage{
2      import metamodel arduino
3      context Block{ covered when ONE instructions iscovered },
4      context If{ covers ^condition },
5      context While{ covers ^condition },
6      //ignore physical-related elements from coverage computation
7      context Board{ ignore (subtypes true) },
8      context Pin{ ignore (subtypes true) },
9      context Module{ ignore (subtypes true) }
10 }
```

Listing 1: Model element coverage rules for the Arduino xDSL

the coverage of the Block elements, we defined a CoverageByContent rule (line 3). It specifies a Block element is covered if at least ONE of its contained instruction elements is covered. In the definition of the Arduino xDSL (Figure 1), the Expression elements are executed once their container Control element is executed. According to this information, we define two CoverageOfReferenced rules (lines 4 and 5) specifying whenever an If or a While element is covered, their condition that is an Expression is also covered. Finally, we define several Ignore rules (lines 7-9) to ignore those elements representing the physical Arduino elements from the element coverage computation.

Our proposed language allows defining several DomainSpecificCoverage instances and importing one in another (using the Import concept of the language). This enables the language engineers to reuse the already defined coverage rules when specifying new coverage rulesets. For example, we defined a second set of coverage rules for the Arduino xDSL for computing branch coverage of the Arduino models. As presented in Listing 2 (line 3), it imports the previously defined coverage rules of Listing 1.

We defined two BranchSpecification rules for the If context with different Conditions (lines 5-12). The first rule applies on the If elements having an elseBlock (line 5) and the second one on those without any elseBlock (line 9). Both rules specify two Branches. Their first branch is the same and is an ExplicitBranch

```
1  ruleset ArduinoBranchCoverage{
2      import metamodel arduino
3      import ruleset ArduinoModelElementCoverage
4      context If{
5          when "self.elseBlock <> null"{
6              branch "self.block",
7              branch "self.elseBlock"
8          },
9          when "self.elseBlock = null"{
10             branch "self.block",
11             branch else
12         }
13     }
14 }
```

Listing 2: Branch coverage rules for the Arduino xDSL

(lines 6 and 10). It determines that the block of the If elements represents an execution branch of the Arduino models. For the second branch, the first rule considers the elseBlock elements as execution branches (line 7) while for the second rule that applies on the If elements without any elseBlock, an ImplicitBranch is defined (line 11).

4.2.2. Executing DSL-Specific Coverage Rules

Given a test case executed against a model that conforms to an xDSL providing a coverage ruleset, by applying the coverage rules, we construct between one and two coverage matrices for the test case: an *element* coverage matrix when there are Inclusion and Exclusion rules, and *branch* coverage matrix when there are BranchSpecification rules.

Algorithm 3 describes the computation. The required inputs are the coverage ruleset provided by the language engineer, the element coverage matrix constructed by the generic coverage computation approach (using Algorithm 2) to be used as the initial element coverage matrix (line 2), and the execution trace of the tested model (it is needed for computing branch coverage).

For each Context of the coverage ruleset (line 4), we repeat the following steps until a fixed point is reached i.e., until the coverage matrix becomes steady (line 5). For each Rule of the Context (line 6), we first select those model elements conforming to its context metaclass and satisfying its Condition (if any), referred to as valid elements (line 7). Next, according to the type of the Rule, different computations should be performed. For example, if the rule is a CoverageOfReferenced (line 8), the coverage status of each valid element is considered as the coverage status of its referenced element (lines 10), and if it is covered, the execution trace will be modified by adding the referenced element after the valid element (lines 11-12). Modifying the trace by adding new covered elements is necessary for branch coverage computation because our proposed approach relies on trace information (details are given subsequently). As another example, if the rule is a BranchSpecification (line 13), we keep the valid elements (lines 14) for computing branch coverage (lines 17).

Algorithm 3: Computing DSL-specific coverage

Input:
coverageRuleset: the DSL-specific coverage rules,
elementsCoverages: the output of Algorithm 2,
trace: list of model elements captured by the execution trace,
Output :
updatedCoverages: a mapping of the model elements and their DSL-specific coverage status

```
1  begin
2      updatedCoverages ← elementsCoverages
3      branchRule_contextObjs ← ∅
4      foreach context ∈ coverageRuleset.contexts do
5          while updatedCoverages changes do
6              foreach rule ∈ context.rules do
7                  validObjs ← elementsCoverages.filter(o | o
                     conformsTo context.metaclass ⋀ o satisfies
                     rule.condition)
8                  if rule is CoverageOfReferenced then
9                      foreach obj ∈ validObjs do
10                         updatedCoverages←(obj.reference,
                             coverage(obj))
11                         if coverage(obj) == COVERED then
12                             trace.addAfter(obj.reference, obj)
13                 else if rule is BranchSpecification then
14                     branchRule_contextObjs←(rule, validObjs)
15                 else if rule is <other rule types> then
16                     ...
17     computeBranchCoverage(branchRule_contextObjs, trace)
```

Branch Coverage Computation. The context elements defined by BranchSpecification rule are roots of a branch. Having this information along with the final execution trace of the model (both provided by Algorithm 3), we compute the branch coverage (see Algorithm 3 line 17). Figure 8 shows some examples of the branch coverage computation on a role model which acts as an example template for discussing the different cases.

We start by checking whether branching root elements are captured in the trace. If the branching root element is not captured, it means none of the contained branches are covered neither their explicit nor implicit branches. In Figure 8(a), the object b1 was not captured by the trace, thus, neither of its branches. However, if there is at least one occurrence of the root element of the branch, then, we check the coverage status of its branches.

As mentioned in Section 4.2.1, a BranchSpecification rule specifies how to detect branches from the root of a branch. In particular, some branches are Explicit model elements and can be identified by running an OCL query on the root element. Of course, an explicit branch is covered if its representative model element is captured in the trace after the occurrence of the root element. Once identified, we compute their coverage status based on their occurrence in the execution trace. Figure 8 (b) shows an example in which the explicit branch i.e.,, object e1, is not covered because its branching model element with a connection to the successor object e1, i.e.,, the explicit branch, are not captured in the trace. The opposite example is illustrated in Figure 8) (c), in which the branching model element as well as the representative model element of the explicit branch are both captured in the execution trace.

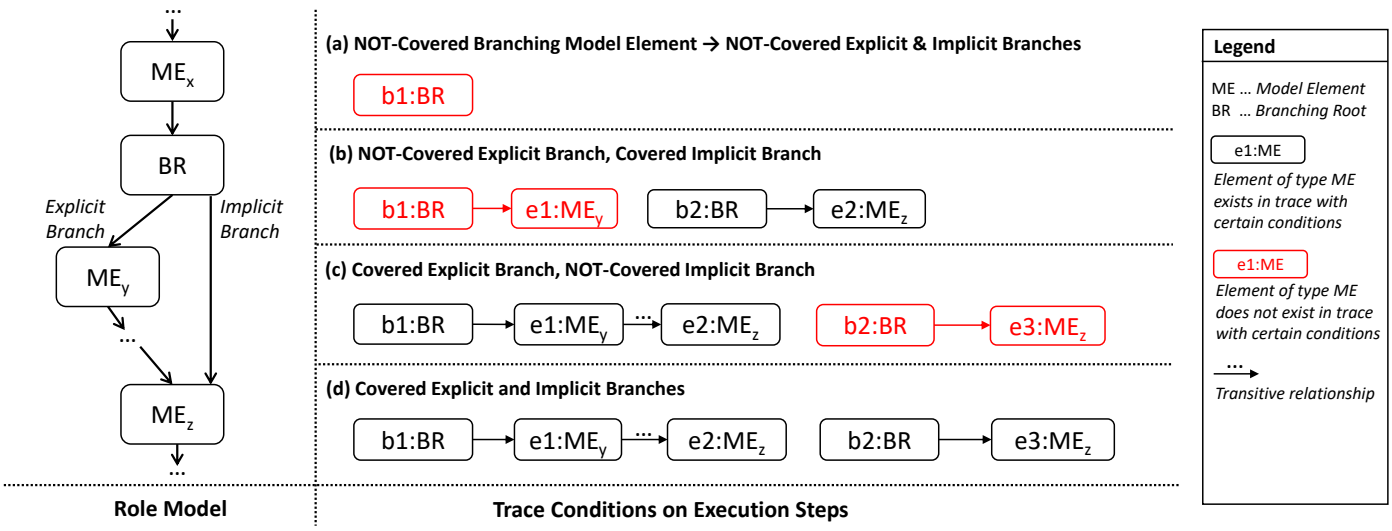

Figure 8: Branch coverage computation for a model based on its execution trace

In addition to explicit branches, we also consider that there might be at most one Implicit branch from a root which has no representative element in the model (such as in a Java program where if statements always originate two branches whether or not an else block comes after). An implicit branch is covered if there is at least one captured root element in the execution trace without a following explicit branch element. We compute their coverage status based on the occurrences of both root elements and the explicit branches in the execution trace. For instance, in Figure 8 (b) the implicit branch is covered i.e.,, element e2 with root object b2. It can be seen as an opposite example in Figure 8 (c), in which the branching model element with a successor connection to the branch merging element is not part of the trace.

After examining all branches, the coverage status of its branching root element is computed. The status of the root element is determined as follows:

- *Not-covered* when all of its branches are *not-covered* (see Figure 8(a)).
- *Partly-covered* when it has both covered and not-covered branches. It can be seen two examples in Figure 8 (labels (b) and (c)).
- *Covered* when all of its branches are *covered*. Figure 8(d) shows an example in which explicit and implicit branches are covered.

It is worth mentioning that, for each set of DSL-specific coverage rules (i.e., each DomainSpecificCoverage instance), we generate a new coverage matrix for the test case.

DSL-Specific Coverage Computation for the Running Example. Considering the running example (the test case of Figure 3 executed against the Arduino model of Figure 2), the result of applying the Arduino-specific coverage rules of Listing 1 is an element coverage matrix constructed as follows. As presented in Table 1: (1) the `button` element is considered as covered after trace analysis (generic coverage), but is then ignored after applying the coverage rule in line 9 of Listing 1; (2) the `if` element is covered based on the trace analysis (generic coverage); (3) the `button==1` Expression does not have any status at first (generic coverage) but it is then considered as covered after running the coverage rule in line 4 of Listing 1 as it defines that coverage of an `if` element implies the coverage of its condition element; (4) the `LED=1` ModuleAssignment is covered based on the trace analysis (generic coverage); (5) the `block` of the `if` element has no status at first (generic coverage) but after running the coverage rule in line 3 of Listing 1, it is considered as covered since its inner instruction (i.e.,, `LED=1`) is covered; (6) the `LED=0` ModuleAssignment is not-covered by the test case (generic coverage); and, (7) the `elseBlock` of the `if` element has no status at first (generic coverage) but since none of its inner instructions (i.e.,, `LED=0`) are covered, it is considered as not-covered (DSL-specific coverage); In the end, the final DSL-specific coverage matrix is equivalent to the content of Table 1 column 3.

Table 1: An excerpt of the DSL-specific coverage computation for the running example which uses the generic coverage computation result (changes in bold)

Object	Type	Generic element coverage	DSL-specific element coverage
button	PushButton	Covered	**No-Status**
if	If	Covered	Covered
button==1	Expression	No-Status	**Covered**
LED=1	Module-Assignment	Covered	Covered
block	Block	No-Status	**Covered**
LED=0	Module-Assignment	Not-Covered	Not-Covered
elseBlock	Block	No-Status	**Not-Covered**

Table 2: An excerpt of the branch coverage matrix for the running example

Branching root	Branches	Branch coverage
if (button == 1)		Partly-Covered
	block	Covered
	elseBlock	Not-Covered
if (infrared sensor == 1)		Covered
	block	Covered
	Implicit branch	Covered

We also executed the coverage rules of Listing 2 for the running example. As it imports the coverage rules of Listing 1, the coverage status of the model's elements will be the one presented in Table 1. In addition, since it has BranchSpecification rules, we computed the branch coverage of the running example as follows. According to Listing 2, there are two BranchSpecification rules for the If Context. The first one applies on the If elements having an elseBlock such as the If element of the Arduino model (Figure 2) checking the level of the button. The second rule applies to the If elements without any elseBlock such as the If element checking the level of the infrared sensor. The result of running the rules on these elements is presented in Table 2. Using the coverage of the Arduino model's elements (Table 1), our proposed approach identified the first If element as partly-covered and the other as covered.

4.2.3. Validating DSL-Specific Coverage Rules

Our proposed language validates the DSL-specific coverage rules concerning their consistency. We define static validation i.e., constraints are validated at compile time and dynamic validation i.e., performed at runtime during the execution of the rules.

Static Validation. Exclusion and Inclusion rules are opposite to each other. The former ignores model elements from the coverage computation while the latter specifies how to compute the coverage of the elements. The proposed language does not allow defining both Inclusion and Exclusion rules for the same Context through the OCL constraint of Listing 3.

```
context Context
def: hasInclusionRule : Boolean = self.rules-> exists(r | r.
     oclIsTypeOf(InclusionRule))
def : hasExclusionRule : Boolean = self.rules-> exists(r | r.
     oclIsTypeOf(ExclusionRule))
inv repeatedContextIncExcCriteria ('The same context cannot have
     both Inclusion and Exclusion rules'):
    not (hasInclusionRule and hasExclusionRule)
```

Listing 3: An OCL constraint for validating Exclusion and Inclusion coverage rules

According to the metamodel of the language (Figure 7), each BranchSpecification rule needs at least one Branch. We explained in Section 4.2.2 that the coverage of an ImplicitBranch is computed based on the coverage of the ExplicitBranches. Therefore, there must be at least one ExplicitBranch and at most one ImplicitBranch in the definition of each BranchSpecification rule, as expressed by the OCL constraint of Listing 4.

```
context BranchSpecification
inv atLeastOneExplicitBranch ('BranchSpecification rule must have
     at least one ExplicitBranch'):
    self.branches -> exists(b | b.oclIsTypeOf(ExplicitBranch))
inv onlyOneImplicitBranch('BranchSpecification rule must have at
     most one ImplicitBranch'):
    self.branches -> select(b | b.oclIsTypeOf(ImplicitBranch))->
     size() <= 1
```

Listing 4: An OCL constraint for validating BranchSpecification coverage rules

Dynamic Validation. As described in Section 4.2.2, we compute the branch coverage of a model using the coverage of its elements. As the Inclusion and Exclusion rules update the coverage status of the models' elements, they must be applied first—before applying the BranchSpecification rules. Moreover, when a Context has several BranchSpecification rules, they must have non-overlapping Conditions. This ensures that each rule will be applied to a unique set of elements, hence avoiding any potential conflict. To perform this validation, we retrieve the valid context elements for each rule (by keeping the context elements satisfying the rule's Condition) and we check if their conjunction is empty.

5. Test Coverage Usage

Among different use cases of utilizing test coverage measurements, this paper focuses on test quality evaluation and model fault localization presented in Sections 5.1 and 5.2, respectively.

5.1. Test Quality Evaluation

A well-known approach for improving test efficiency is improving the quality of the tests by considering a criterion such as test coverage. Usually, the higher the coverage is for a test suite implies that the test suite exercises more parts of its model under test, so has a higher quality.

To perform quality evaluation for a given test suite, a coverage percentage must be calculated for each of its computed coverage matrices. In this paper, we compute the coverage percentage for each supported metric as follows:

$$\textit{Element coverage \%} = \frac{\#\ of\ covered\ elements}{\#\ of\ traceable\ elements}$$

$$\textit{Branch coverage \%} = \frac{\#\ of\ covered\ branches}{\#\ of\ traceable\ branches}$$

As can be seen, the not-traceable elements (i.e., the ones with NO-STATUS coverage status) must be reduced from the total

number of elements when calculating the coverage percentage. For example, the coverage percentages for the running example (having a total of 35 elements) are:

- model element coverage computed by the generic approach: 14/19 = 73.68 %
- model element coverage computed by the DSL-specific approach: 16/18 = 88.89 %
- model branch coverage computed by the DSL-specific approach: 3/4 = 75 %

5.2. Model Fault Localization

When a test case fails, it is unfortunately often challenging to localize the defect causing the failure. In order to provide dedicated support for this step, a multitude of fault localization techniques have been proposed in the past such as SBFL which is based on utilizing the computed coverage measurements of different test executions [11]. As our proposed model coverage metrics are generic with respect to the supported xDSLs, it allows us to offer SBFL for any xDSL as well.

In the realm of software testing, SBFL is usually applied at the statement level, i.e., it uses the statement coverage of the tested program and calculates the suspiciousness of each statement of the program [11]. In this work, we adapt SBFL for xDSLs by substituting the notion of statement with the generic concept of *model element*. Accordingly, considering a test suite of a model, our proposed *Model Fault Localization* component uses the execution result and the *model element coverage* matrix of the test suite (i.e., generated by our *Model Coverage Computation* component) to calculate the suspiciousness-based ranking of the model's elements using SBFL techniques.

In the current state-of-the-art SBFL, each SBFL technique introduces a dedicated formula that is based on a set of values (note that we adapted them for the context of models) which are computed from the test results and coverage information:

- NCF: number of failed test cases that cover the element
- NUF: number of failed test cases that do not cover the element
- NCS: number of successful test cases that cover the element
- NUS: number of successful test cases that do not cover the element
- NS: total number of successful test cases
- NF: total number of failed test cases
- NC: total number of test cases that cover an element
- NU: total number of test cases that do not cover an element

For instance, a well-known formula is Tarantula [29] which is $(NCF/NF)/(NCF/NF + NCS/NS)$. It is indeed based on two main assumptions: (*i*) elements which are executed by more failed test cases are more likely to be faulty, and (*ii*) the ones which are executed by more passed test cases are less likely to have a fault. In the current implementation of our approach, we support 18 existing formulas. The details on the realization of each formula can be found in the Appendix A.

It should be noted that since our proposed model fault localization technique is currently solely based on SBFL, it is not able to detect faults due to missing elements as it is a shortcoming of the SBFL approach (as already mentioned in Section 2.5). For localizing those kinds of faults, other techniques must be applied, which is left to future work.

6. Tool Support

To provide tool support of our proposed approach for a given xDSL, we need: (1) an API to manipulate the abstract syntax and the execution semantics of the xDSL (i.e., needed for the xDSL analysis step); (2) an API to manipulate the conforming models; (3) facilities to execute the models and produce their execution traces; and (4) an API to manipulate the execution traces to deduce what was covered. Considering the *genericity* objective regarding the supported xDSLs, we needed a language and modeling workbench that support all these features for any xDSL. As Eclipse GEMOC studio meets these expectations, we implemented a prototype of our proposed framework for the Eclipse GEMOC Studio [13]. However, the prototype is potentially adaptable (with some effort) to another environment that can provide all the mentioned characteristics.

To address the need for an existing testing framework, we used our previous work [9] which itself uses a generic execution trace management for xDSLs proposed by Bousse et al. [16, 17]. All the components of the framework (the *Model Coverage Computation*, the *Model Fault Localization*, and the *Analysis Tools* components as shown in Figure 4), are implemented in Java and are connected using the Eclipse extension point mechanism. In particular, for the proposed language, its abstract syntax is defined using an Ecore metamodel, its textual concrete syntax is implemented as an Xtext grammar along with scope providers and static validators, and its execution semantics is implemented in Java.

The suspiciousness computation implementation is based on a previous work provided by Troya et al. [12] for fault localization in model transformations. We have adapted their work for general model elements in order to support any xDSL. In the current realization, our tool supports 18 SBFL techniques but adding new ones is possible in our framework by dedicated means. By taking a look at the literature, there are approximately over 30 SBFL techniques [20, 11, 30] available. What they have in common is that they all use the same set of variables, as explained in Section 5.2 (i.e.,, NCF, NCS, NS, NF), to compute the suspiciousness-based ranking. This means that any existing formula that is based on the aforementioned variables can be added to the framework.

Figure 9 shows a screenshot of our tool running in the Eclipse GEMOC Studio modeling workbench, after executing a test suite against the running example. The source code is accessible from a Zenodo repository [31]. In the project explorer on the left, there are two projects, one containing the Arduino model (shown in Figure 2) and another containing a test suite written for it using our testing framework [9]. The constructed coverage matrices can be persisted as XMI files conforming to the format presented in Figure 5 upon the request of the user—the user can select the related options in the run configuration. For example, Label 1 in Figure 9 indicates the generated coverage files.

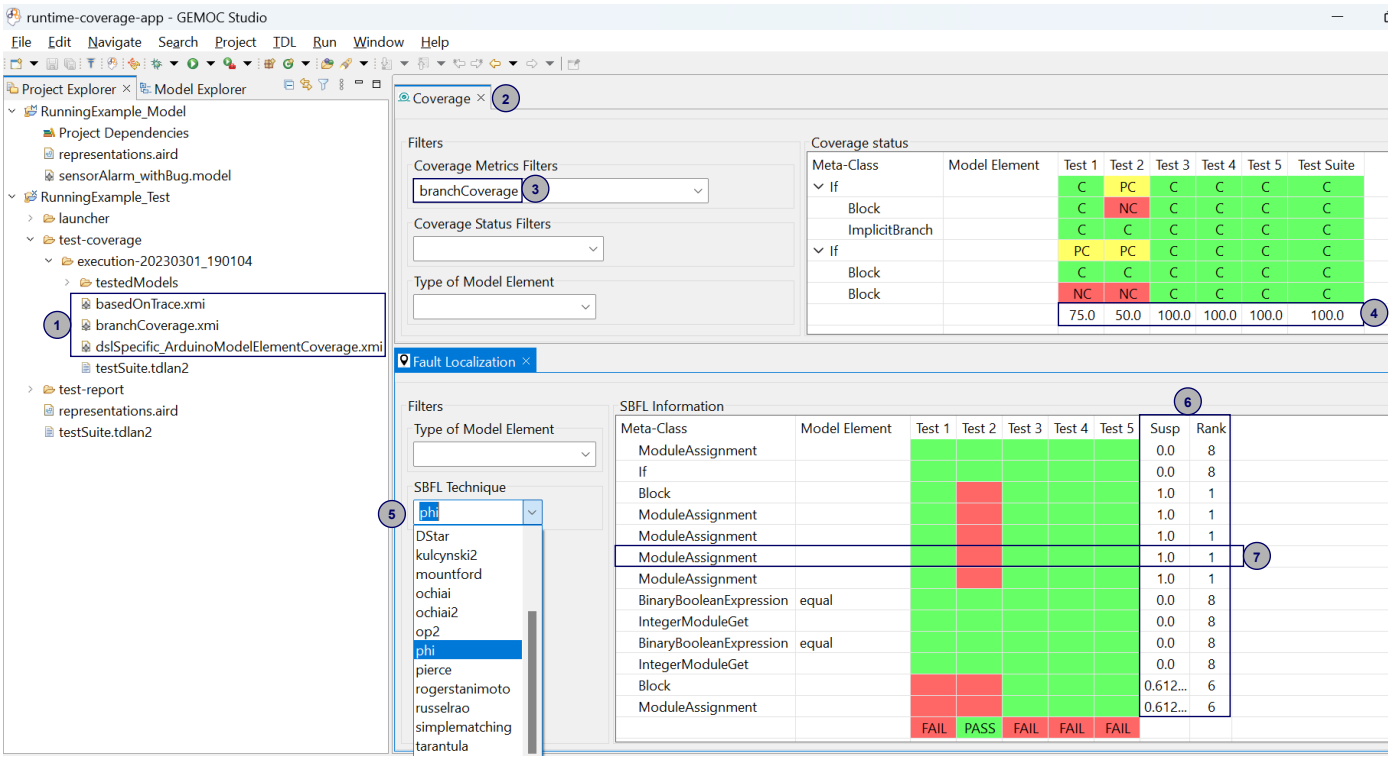

Figure 9: Screenshot of the provided tool support based on the Eclipse GEMOC Studio showing the coverage and the fault localization views for the running example of this paper.

We also provided a graphical view (label 2) to display the coverage measures computed for the test cases and for their test suite (at the center) based on a selected coverage metric (label 3). In the *Coverage Metrics Filters*, there is one entry for the generic model element coverage computed based on the execution traces, and one entry per coverage ruleset of the xDSL that the model under test conforms to. For example, as the Arduino xDSL has a coverage ruleset for computing branch coverage (presented in Listing 2), we can select the corresponding filter (label 3) and see (*i*) the coverage status of both the branching root elements and their branches—green for COVERED, red for NOTCOVERED, and yellow for PARTLYCOVERED status—and (*ii*) the coverage percentage at the last row (label 4) for the running example. The user can also use two other filter options, one to find all the elements with a specific coverage status (*Coverage Status Filters*), and another to find the coverage status for a specific type of the elements (*Type of Model Element* filter).

To run SBFL on the tested model, we developed a tool titled "Fault Localization" providing a graphical interface for the user (on the bottom). It shows the traceable elements of the tested model, their coverage status by each test case, and the test execution result (in the last row). 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 users in debugging their models by providing direct links to the locations of the faulty elements.

For instance, if we chose Phi as the concrete SBFL technique, the tool calculates the first rank for the third ModuleAssignment of the second if condition of the Arduino model of Figure 2 where the defect is located (label 7). For this case, the first ranked element is the faulty element, thus the rank for the faulty element is correctly calculated. However, please note that there are also other elements with the same rank. This is a common result of existing SBFL techniques, due to the so-named *tied* elements [11]. One common tie-breaking strategy for software programs is ordering based on line numbers in a text editor [11] and we support ordering based on model element position in a tree editor (i.e., provided by Ecore model editor).

7. Evaluation

In this section, we report on the performed empirical study to evaluate our proposed framework. In particular, we aim to answer the following research questions:

RQ1: How much parameterization is required by our framework to realize the intended coverage computation for selected xDSLs?

RQ2: To what extent our approach can reproduce the results of existing coverage computation approaches?

RQ3: What is the performance of existing SBFL techniques when they are used to localize faults in executable models considering the computed element coverage measurements?

7.1. Experiment Setup

We describe in the following the setup we considered for answering each research question.

Table 3: Evaluation setup data at a glance

		xFSM	xArduino	xPSSM	xMiniJava	Total
xDSL	Abstract syntax size (n. of EClasses)	3	59	39	76	-
	Execution semantics size (LoC)	111	768	975	1,042	-
Tested models	Number of tested models	5	6	4	6	21
	Size range of tested models (n. of EObjects)	7-133	18-59	61-154	31-571	7-571
Test artifacts	Total number of test cases	49	22	153	77	301
	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	1,252
	Number of killed mutants	194	457	308	120	1,079

Setup for RQ1. For RQ1, we investigate whether the proposed framework can be parameterized for different xDSLs without considerable effort. Accordingly, we chose four existing xDSLs from different domains showing different characteristics:

- *xFSM*: A small language for developing Finite State Machines for processing String values.
- *xArduino*: A language for simulating Arduino boards and their execution logic (we already described this language in Section 2.1).
- *xPSSM*: A partial implementation of the Precise Semantics of UML State Machines (PSSM) [32] which supports modeling of discrete event-driven behavior.
- *xMiniJava*: An implementation of a Java subset based on the MiniJava project [33], allowing the definition of simple Java programs that can be directly executed by an execution engine in addition to the JVM. Note that it is not a typical xDSL and is defined just for experimental purposes of this evaluation which also allows a comparison with mature tools as we will see in the following.

As presented in Table 3, the considered xDSLs have different sizes as the number of classes specifying their abstract syntax and the number of Lines of Code (LoC) of their execution semantics. On average, we defined 5 differently sized models using each xDSL, ranging from 7 to 571 number of objects. In addition, using an existing testing framework [9] in the Eclipse GEMOC Studio, we defined a set of test cases per model, altogether, 301 test cases for 21 models. The number of test cases for each model ranges from 3 to 81.

Concerning the intended coverage computations for the selected xDSLs, we have to note that unfortunately there is no "gold standard" available to which we can compare, except for xMiniJava for which mature coverage tools are available. The latter aspect is particularly focused on answering RQ2. Thus, for the other xDSLs, the authors have discussed and evaluated different coverage options and selected a commonly agreed intended coverage for the given xDSLs and provided models.

For answering RQ1, we select as dedicated metrics the number and types of the required coverage rules for achieving both, element and branch coverage. To provide an estimate of the effort required to develop these rules, we also measure the Lines of Code (LoC) necessary to develop the DSL-specific coverage support for the chosen languages.

Setup for RQ2. For answering RQ2, we compare our coverage computation component with existing code coverage tools. As xMiniJava is a Java-based 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. This allows us to compare our coverage computation approach with existing well-known Java coverage tools. For this comparison, we have chosen JaCoCo [34], CodeCover [35], and OpenClover [36] as they are open-source coverage tools supporting JUnit tests of Java programs. Among different code coverage metrics supported by them, we use *statement coverage* as it is the closest to our model element coverage metric, as well as *branch coverage*.

We reused the Java programs provided by the MiniJava project [33]. For the comparison with JaCoCo, we had to perform two minor modifications on them. First, as some of the closed brackets that are in a separate line of code are counted by JaCoCo when computing statement coverage (e.g., the closing curly bracket of if statements), we put them next to their last precedent statement. Second, we transformed compound expressions of the conditional elements (e.g., `if(a && b){}`) to a set of atomic expressions (e.g., `if(a){if(b){}}`) to make the JaCoCo branch coverage metric similar to a decision coverage metric [37]. Finally, we transformed the test cases of xMiniJava models—according to Table 3, 77 tests for 6 xMiniJava models—to JUnit tests for equivalent Java programs.

In each performed comparison, we investigate whether the coverage status of each Java statement/branch is the same as for its equivalent xMiniJava element/branch. In case there are deviations, we document these deviations and investigate if there is a parameterization option in our proposed framework, i.e.,, a dedicated coverage rule, which counteracts the deviation.

It has to be noted that we could not find any existing coverage computation tool for the other xDSLs. Consequently, our evaluation for RQ2 is limited to xMiniJava. However, we would like to emphasize that for this xDSL, we have the opportunity to compare against mature coverage tools.

Setup for RQ3. RQ3 targets the utilization of the model element coverage measurements for subsequent fault localization activities. As our proposed fault localization approach relies on SBFL techniques, we answer this question from two viewpoints. First, we need to assess whether our fault localization approach correctly localizes the faulty element of a model i.e., gives it

the first rank among all the elements of the model. Second, as there are plenty of SBFL formulas for suspiciousness ranking measurement—18 of which are currently supported by our tool—we aim to evaluate the effectiveness of each formula for the context of executable models.

Answering RQ3 first requires running the fault localization component on each considered model. To do this, there must be at least one defect in the model and at least one of its related test cases must be failing. A well-known technique for producing faulty programs is mutation analysis, 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. For the scope of this evaluation, we used WODEL [38], a generic mutation analysis framework that provides facilities to define mutation operators for an xDSL, then it automatically generates mutants for the models conforming to that xDSL.

As Table 3 presents, we defined in total 184 mutation operators for our considered xDSLs. Thereafter, WODEL generated a total of 1252 mutants for our 21 models. 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 fails it. Among 1252 mutants, 1079 of them were killed by our written test suites. To know the exact location of their injected fault, we used a tool named EMF Compare [39] to automatically find the faulty element of each mutant by comparing it with the original model.

For effectiveness measurement of SBFL techniques, there is a well-known metric in the literature called *EXAM score* [40]. It measures the percentage of the elements that have to be examined to reach the first faulty element of the model (a value in the range (0,1]):

$$EXAM\ Score = \frac{\#\ of\ examined\ elements}{\#\ of\ all\ elements}$$

Therefore, an SBFL formula with a lower EXAM score is considered more effective in fault localization. However, SBFL formulas may calculate the same suspiciousness score for several elements, hence tying them to the same position in the ranking. Therefore, one strategy for effectiveness measurement is considering the Best Case (BC), the Average Case (AC), and the worst case (WC) scenarios [11]. In the BC and WC scenarios, the faulty element is inspected first and last in the tie, respectively. For example, considering the xArduino model of Figure 2, the total number of elements in the model element coverage matrix is 18 (as mentioned in Section 5.1). If an SBFL formula correctly gives the first rank to the faulty element, but also to 3 other elements (i.e., 4 elements in a tie), the BC EXAM score is $\frac{1}{18} = 0.055$, the WC EXAM score is $\frac{4}{18} = 0.222$, and the AC EXAM score is $\frac{2}{18} = 0.111$. It means to find the faulty element of the model, 5 % of its elements must be examined in the best case, 22 % in the worst case, and 11 % in the average case. In the presented subsequent analysis, we measure EXAM scores for all BC, WC, and AC scenarios.

Evaluation data. We published our evaluation data in a Zenodo repository [31].

7.2. Evaluation Result

In the following, we present the evaluation results to answer the aforementioned research questions. In particular, we present the main results summarized in a respective table for each research question.

Answering RQ1. In the first research question, we aim to evaluate the effort for using the proposed framework for a set of selected xDSLs. Specifically, we are questioning the level of required parameterization for each xDSL to realize the intended coverage measurements for their conforming models. The overall result of our evaluation is presented in Table 4.

After running the test cases on the available models—a total of 301 test cases on 21 models—we observed that the generic coverage computation approach successfully computed the element coverage for all the test cases. Nevertheless, the computed element coverage for the xArduino, xPSSM, and xMiniJava models was limited due to the reasons already mentioned in Section 4.1.3. To lift the limitations, we defined a set of Inclusion and Exclusion coverage rules for their corresponding xDSLs, totally 17 and 9 rules, respectively. We already explained some of the rules defined for the xArduino in the Listings 1 and 2 of Section 4.2.1. For xPSSM and xMiniJava, several rules were defined to infer the coverage of a container element from the coverage of its contained elements (using CoverageByContent Inclusion rule type). For example, in xPSSM models, a State element is covered if at least one of its outgoing Transition elements is covered and a Class element in an xMiniJava model is covered if any of its contained elements are covered. We further explain the coverage rules for xMiniJava later in providing the answer to RQ2.

Moreover, as all the considered xDSLs have the notion of branching in their definition, we instantiated a set of BranchSpecification rules for each of them (a total of 9 rules). Interestingly, for both state machine-based languages, only one branch specification rule was necessary because execution branching occurs only from States with more than one outgoing Transition. For xArduino and xMiniJava we needed to define 3 and 4 rules, respectively, because they have several branching concepts (i.e., If, While, and For elements) with the possibility of having Implicit branches (i.e., If elements without else blocks).

It is also worth mentioning that, in some cases, we needed to define new Inclusion and Exclusion coverage rules to be able to compute branch coverage. For example, as discussed in Section 4.2.1 for the xArduino, the Block elements contained by If elements represent explicit branches, thus their coverage status is needed to compute also the xArduino branch coverage. However, the generic coverage measurement (i.e., without applying any coverage rule) was not able to set any coverage status for the Block elements. Using MoCL, we were able to define a coverage rule to infer their coverage status based on their contained Instruction elements (as shown in Table 1).

Altogether, we have implemented 35 coverage rules consisting of altogether 98 LoC for 4 xDSLs using MoCL. Through the application of this set of coverage rules, we successfully computed the intended *model element coverage* and the *model branch coverage* for all considered test cases. Thus, we conclude

Table 4: Evaluation result at a glance

			xFSM	xArduino	xPSSM	xMiniJava	Total
#of tests with computed coverage			49	22	153	77	301
DSL-Specific Coverage	coverage rules	#of Inclusion rules	0	5	10	2	17
		#of Exclusion rules	0	3	0	6	9
		#of Branch Specification rules	1	3	1	4	9
		Total #of rules	1	11	11	12	35
		DSL-specific coverage size (LoC)	5	31	28	34	98
	Model element coverage %		100%	100%	100%	97.5%-100%	-
	Branch coverage %		100%	100%	100%	91.7%-100%	-
SBFL	#of fault localized mutants		194	452	304	119	1,069

that an efficient realization of the intended coverage computation for xDSLs is possible with MoCL, supporting both model element coverage and model branch coverage.

Answering RQ2. This research question targets the reproducibility of specifically developed coverage computation approaches by our proposed *Model Coverage Computation* component. To answer this question, we compared the coverage matrices generated by our proposed component for the xMiniJava tests with those generated by the three considered code coverage tools for the equivalent JUnit tests. A summary of our comparison is presented in Table 5.

According to our investigation, each of the considered coverage tools has its own considerations for computing coverage, including (*i*) Java classes are considered as statements by JaCoCo but not by CodeCover and OpenClover; (*ii*) contrary to JaCoCo and OpenClover, CodeCoder does not count conditional and looping statements (e.g., if, for, and while) when computing statement coverage [41]; and (*iii*) the body of a looping statement is not counted as a branch by CodeCover [42] while it is considered by JaCoCo and OpenClover.

Comparing our model element coverage measurements with statement coverage results produced by each tool, we achieved the same result as:

- JaCoCo when applying a CoverageByContent rule for the class elements;
- CodeCover when applying Ignore rules for the if, for, and while elements;
- OpenClover without any need for model element coverage rules.

Moreover, we achieved the same result for the branch coverage measurements with:

- JaCoCo and OpenClover when applying BranchSpecification rules for the looping elements (i.e., for and while elements);
- CodeCover when not considering the BranchSpecification rules for the looping elements.

In each performed comparison, we manually verified that the coverage status of each Java statement/branch by each JUnit test is the same for its equivalent xMiniJava element/branch by its related test, meaning that our approach provides the same result for the user. This result demonstrates that our approach can reproduce the measurements of different coverage tools by defining a few coverage rules for the xMiniJava DSL using the MoCL, hence showing the flexibility of our parametrization approach.

Answering RQ3. To answer this research question, we aim to evaluate the utilization of our model element coverage metric for fault localization tasks. For this, we first have to investigate whether our *Model Fault Localization* component correctly finds the faulty element of each mutant. Thus, we executed the supported SBFL techniques (18 techniques as presented in Appendix Appendix A) on the 1079 killed mutants and checked the rank of the mutants' faulty element calculated by each SBFL technique. We observed that for 1069 examined mutants (i.e., 99 %), there is at least one SBFL technique supported by our framework that calculated the rank of its faulty element as first. This result emphasizes the general usefulness of our model element coverage measurement as a foundation for SBFL techniques.

Second, we calculated the EXAM score of each supported SBFL technique to assess their effectiveness in the context of localizing faults in executable models. Table 6 shows EXAM scores of each technique in the BC, AC, and WC scenarios for the considered xDSLs both individually and altogether. Overall, it is evident that the AC EXAM score of most of the SBFL techniques is between 0.25 and 0.35. Some have AC EXAM scores above 0.35 up to 0.66 which indicates a low performance for localizing the actual faulty element.

As can be seen in Table 6, *Arithmetic mean* [40] is performing well in the BC scenario, with 0.038 average EXAM score, but it does not perform well in the AC and WC scenarios relative to other techniques. This technique results in many ties in the ranking, which harms the AC and WC scenarios. As we see for all investigated techniques, there are relatively large spans between the BC and WC, meaning that the problem of having many ties of SBFL techniques [43] also occurs for executable models. The most effective approaches when considering all the scenarios and xDLS are in order (1) *Rogers & Tanimoto* [44] and *Simple matching* [11], with the third lowest BC score (0.113) and

Table 5: Reproducing three Java code coverage tools for the xMiniJava DSL

Reproduction of	**JaCoCo**	**CodeCover**	**OpenClover**
statement coverage by model element coverage	✓*with* a CoverageByContent rule for Class elements	✓*with* Ignore rules for conditional, and looping elements	✓*without* any rule for Class, conditional and looping elements
code branch coverage by model branch coverage	✓*with* BranchSpecification rules for looping elements	✓*without* BranchSpecification rules for looping elements	✓*with* BranchSpecification rules for looping elements

Table 6: EXAM scores (AC, BC, and WC values) for investigated SBFL techniques: language-specific and overall performance results.

	xFSM			xArduino			xMiniJava			xPSSM			Overall		
Technique	AC	BC	WC	AC	BC	WC	AC	BC	WC	AC	BC	WC	AC	BC	WC
Arithmetic Mean	0.387	**0.074**	0.699	0.318	**0.066**	0.628	0.340	0.057	0.622	0.392	**0.013**	0.772	0.333	**0.038**	0.627
Barinel	**0.203**	0.086	**0.322**	0.314	0.140	0.569	0.391	0.145	0.638	0.214	0.080	0.348	0.246	0.079	0.414
Baroni et al.	0.448	0.321	0.576	**0.241**	0.077	**0.428**	0.236	0.092	0.379	0.420	0.234	0.606	0.306	0.147	0.465
Braun-Banquet	0.449	0.322	0.577	0.253	0.070	0.445	0.233	0.088	0.378	0.428	0.239	0.616	0.317	0.149	0.485
Cohen	0.279	0.173	0.385	0.295	0.101	0.569	0.262	0.088	0.435	0.307	0.185	0.430	0.265	0.111	0.419
DStar	0.550	0.412	0.687	0.504	0.286	0.643	0.363	0.216	0.510	0.458	0.271	0.646	0.405	0.244	0.566
Kulcynski2	0.446	0.318	0.573	0.251	0.070	0.445	0.228	0.083	0.373	0.426	0.239	0.613	0.314	0.147	0.481
Mountford	0.494	0.347	0.641	0.455	0.231	0.643	0.288	0.108	0.468	0.427	0.238	0.616	0.384	0.179	0.588
Ochiai	0.446	0.319	0.574	0.250	0.070	0.445	0.230	0.085	0.375	0.424	0.237	0.611	0.314	0.147	0.481
Ochiai2	0.407	0.219	0.594	0.308	0.099	0.599	0.273	0.090	0.456	0.392	0.165	0.620	0.322	0.115	0.529
Op2	0.384	0.320	0.499	0.262	0.094	0.444	**0.221**	0.080	**0.361**	0.348	0.266	0.429	0.288	0.172	0.403
Phi	0.278	0.172	0.384	0.295	0.101	0.569	0.261	0.087	0.435	0.307	0.184	0.430	0.265	0.111	0.419
Pierce	0.731	0.525	0.937	0.576	0.265	0.918	0.635	0.319	0.950	0.680	0.482	0.879	0.655	0.394	0.915
Rogers & Tanimoto	0.311	0.225	0.398	0.260	0.082	0.448	0.274	0.120	0.428	**0.209**	0.120	**0.299**	**0.242**	0.113	**0.371**
Russel-Rao	0.515	0.312	0.718	0.392	0.081	0.668	0.279	**0.053**	0.504	0.480	0.255	0.703	0.413	0.165	0.661
Simple Matching	0.311	0.225	0.398	0.260	0.082	0.448	0.274	0.120	0.428	**0.209**	0.120	**0.299**	**0.242**	0.113	**0.371**
Tarantula	0.478	0.323	0.633	0.264	0.100	0.503	0.277	0.101	0.453	0.426	0.218	0.634	0.324	0.142	0.506
Zoltar	0.446	0.318	0.573	0.251	0.070	0.445	0.228	0.083	0.373	0.427	0.240	0.613	0.324	0.142	0.506

the lowest AC (0.242) and WC (0.371) scores; (2) *Barinel* [45] with the second lowest BC (0.079) and AC (0.246) scores, and the third least DC (0.414) score; (3) *Phi* [46] and *Cohen* [47] with slightly higher BC (0.111), AC (0.265), and DC (0.419) scores; and (4) *Op2* [47] which although is the fourth most effective approach in the BC scenario (0.172), has a reasonable AC (0.288) and the second least WC (0.403) scores.

Moreover, there are also less performant approaches on the overall level, respectively, *Pierce* [11], *DStar* [48], *Russel-Rao* [49], and *Mountford* [50] because of having very high EXAM scores in all the scenarios.

For the rest of the approaches (i.e., *Ochiai2* [51], *Tarantula* [29], *Ochiai* [52], *Baroni-Urbani & Buser* [50], *Kulczynski2* [47], *Zoltar* [53], and *Braun-Banquet* [11]), the EXAM scores are almost in the same range in all the scenarios; BC = [0.115, 0.149], AC = [0.306, 0.324], WC = [0.465, 0.529]. In comparison to other techniques, they presented a medium level of effectiveness for model fault localization.

Interestingly, for different xDSLs we have different winning techniques such as *Barinel* for xFSM, *Baroni et al.* for xArduino, *Op2* for xMiniJava, and *Rogers & Tanimoto* as well as *Simple matching* for xPSSM. This shows that depending on the given xDSL, it makes sense to use particular SBFL techniques for fault localization.

7.3. Threats to Validity

In this subsection, we identify threats to validity according to the four main categories defined by Wohlin et al. [54] and how we mitigated them as follows.

Construct Validity. The validity of our coverage computation approaches was compared with three existing well-known coverage tools. Nevertheless, there are still other tools available such as Cobertura [55] which can be used for further comparisons and we left this as future work. However, considering three existing approaches which are used in practice should give enough evidence that the validity for the investigated cases is given. As we did not have a pre-existing gold standard for the coverage information for the used executable models, we developed our own reference understanding about intended coverage measurements. We performed several iterations and used discussions in the author team to come to a common understanding. Furthermore, we also took inspiration from the existing coverage tools for Java programs. However, we cannot make any claims that someone may not have a different understanding of coverage measurements for the given xDSLs and used models. With the parametric approach, we have also a mechanism to reflect different understandings by developing new or customizing existing coverage rules for the four used xDSLs.

Internal Validity. As SBFL is incapable of locating bugs that are caused by missing code, we did not make use of mutation operators that define faults as the removal of existing model elements. This threatens the internal validity of our study. However, to overcome this threat, extensions of our framework with additional fault localization techniques which can deal with missing

elements are required as future work. Furthermore, we have performed minor code modifications in order to deal with concrete textual syntax peculiarities of Java code coverage tools. To mitigate any negative impact such as changing the semantics of the programs, we carefully performed the modifications and did regression testing using the available test cases.

External Validity. In evaluating the parameterization effort for our framework, we only considered four languages, so there is an external threat that the framework might not work as expected for other xDSLs. However, we aimed to use different kinds of languages ranging from well-known modeling standards (xPSSM) over specific modeling languages for particular hardware (xArduino) to more general programming languages (xMiniJava). However, we cannot state any conclusions which go beyond the paradigms of the used xDSLs. Also, we assessed the reproducibility of our framework by only considering xMiniJava as we were not able to find relevant coverage computation tools for other xDSLs which we could compare to. Additionally, we defined our framework on top of the Eclipse GEMOC Studio as a reference for xDSL implementations. As there are other language workbenches [19] available, additional studies are required to validate the portability of our proposed approach.

Conclusion Validity. By investigating RQ3, we observed the performance of SBFL techniques in localizing the faulty element of models and we did a comparison between them based on the well-defined EXAM score in the AC, BC, and WC scenarios. However, it is still an open question whether there is a relationship between the context of an xDSL and the most effective SBFL technique for the fault localization of its conforming models. Nevertheless, such an analysis requires a more detailed empirical evaluation. Currently, we can only provide some recommendations based on the specific results we got for the different xDSLs, used models, and applied mutations to simulate the faults.

8. Related Work

With respect to the contributions of this paper, we discuss three threads of related work. First, we discuss existing work on model coverage. Second, we elaborate on frameworks allowing the development or customization of the coverage tools. Finally, we elaborate on existing approaches for fault localization in models. We also acknowledge that there are related fields such as compiler testing [56, 57] or testing of interpreters [58] which may be complementary used for testing the implementation of xDSLs. However, in the scope of this work, we only focus on testing the executable models realized in xDSLs.

8.1. Model Coverage Approaches

Several research efforts have proposed the usage of existing coverage techniques for specific modeling languages, e.g., see logic coverage for State Machines [59], data-flow coverage for executable UML models [60], branch coverage for activity diagrams [61], among many others. However, to the best of our knowledge, there is no generic coverage criterion for models. Also, this topic is not yet discussed within the context of language workbenches [19].

8.2. Customizable Code Coverage

Some approaches focus on the use of program transformations to ease the construction of code coverage tools. The approach presented in [62] is about the specification of test probes, using the *Rule Specification Language* provided by Semantic Designs[3] (DMS). The presented approach is exemplified by implementing branch coverage rules for C programs. Following this idea, Yue & Gray [63] propose a DSL called SPOT (Specifying PrOgram Transformation). The goal of SPOT is to reduce complexity in defining source-to-source transformations. The feasibility of the approach is demonstrated by implementing statement coverage and branch coverage of C programs. These two aforementioned approaches ease the program transformation to install the test probes, but additional tooling is needed such as a visited vector as well as an engine to collect the coverage results. Our tool does not need additional tooling—by default we provide views to visualize the coverage information—but also provide customization points through a dedicated DSL for defining language-specific coverage measurements.

Other proposals are related to the implementation of test coverage frameworks [64, 65, 66]. For example, Misurda et al. [64] propose a tool called *Jazz* for testing Java programs and computing coverage measures based on the corresponding execution paths. Bordin et al. [65] introduce their tool *Couverture* which is able to measure structural coverage by providing a virtualized execution platform. Sakamoto et al. [66] propose an extensible tool called *Open Code Coverage Framework* (OCCF), which supports a set of code coverage criteria. In addition, OCCF supports the addition of new code coverage metrics as well as customization for new programming languages. OCCF targets 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 models.

A broad range of existing code coverage tools exists for Java applications, such as JaCoCo [34], OpenClover [36], CodeCover [35], or Cobertura [55]. To the best of our knowledge, these tools mostly lack mechanisms to add new languages except OpenClover [67] and CodeCover [68]. The OpenClover tool has an extensibility mechanism, but the precondition to use it is that the new programming language must be somehow mapped to the Java/Groovy-like program structure (i.e.,, class files, methods, and statements). Therefore, languages that do not fulfill this requirement, cannot benefit from this tool. In the case of the CodeCover tool, the new language must provide a parser based on JavaCC [69], and implement a set of interfaces. This is a similar approach to ours, but we provide the language engineer with a compact DSL and dedicated tool support.

In summary, current approaches for coverage computation are mainly defined for GPLs. In this sense, with our framework, we aim to fulfill the need for generic approaches for DSLs by proposing new coverage metrics and measurement techniques

[3] `www.semdesigns.com`

that are adaptable to DSLs. In particular, we propose two generic coverage metrics, including model element coverage and model branch coverage. Moreover, our framework can be customized for a given DSL by using our dedicated model coverage language. The resulting coverage computation support can be used in subsequent steps and in this paper, we applied it for test quality evaluation and model fault localization.

8.3. Fault Localization for Models

There are several research efforts proposing the application of SBFL to specific modeling languages [21]. For instance, some studies detect the faulty element in model transformations [12, 70]. Troya et al. [12] 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. [70] propose an optimization strategy of SBFL by adding weight values to the test models as well as statistical coverage information. Raselimo & Fischer [71] present the usage of SBFL methods for context-free grammars based on a modified parser that 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 fault localization in model transformations, some approaches are aiming for finding faulty elements in models. Wang et al. [72] propose the application of fault localization techniques for declarative models implemented in Alloy. Another line of research aims for detecting errors in models with the use of evolutionary algorithms. BLiMEA [73] and Ebro [74] detect errors in models with the help of evolutionary algorithms. Arcega et al. [75] compare these proposed tools for bug localization and show that the combination of these tools outperforms existing approaches. A very recent work presented in [76] aims for specific support for fault localization by using evolving simulations to locate bugs in models of video games. However, none of these mentioned approaches consider the execution semantics of modeling languages to detect errors or the support of multiple languages, thus, our approach can be considered complementary to the others, and vice versa.

9. Conclusions and Future Work

In this paper, we proposed a novel language-parametric test coverage framework for xDSLs. It offers (*i*) two generic coverage metrics, including model element coverage and model branch coverage; (*ii*) a language allowing language engineers to parameterize coverage metrics for a given xDSL; (*iii*) generic techniques for computing model coverage based on the proposed metrics; and (*iv*) two applications of the coverage measurements including test quality evaluation and automatic fault localization. In our evaluation, we observed that an automated and language-parametric framework for coverage computation enriches the DSL definitions with further V&V techniques at a reasonable cost.

In future work, we consider the following lines of research: defining new coverage metrics such as path coverage, providing further support for the definition of DSL-specific coverage rules, e.g.,, by analyzing the execution semantics, adopting other coverage-related testing techniques, such as test case generation and test amplification, and developing optimized SBFL techniques for specific xDSLs. We also consider to investigate how to import traces generated with other interpreters, e.g.,, pure Java/EMF-based interpreters. In addition, we will explore how our presented concepts can be transferred to other language engineering approaches, potentially following different computing paradigms such as term rewriting as used in the K framework [77].

Acknowledgement

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

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

[3] D. Lübke, T. van Lessen, Bpmn-based model-driven testing of service-based processes, in: Enterprise, Business-Process and Information Systems Modeling, Springer, 2017, pp. 119–133.

[4] J. Iqbal, A. Ashraf, D. Truscan, I. Porres, Exhaustive simulation and test generation using fuml activity diagrams, in: P. Giorgini, B. Weber (Eds.), Advanced Information Systems Engineering, Springer, 2019a, pp. 96–110.

[5] H. Wu, J. Gray, M. Mernik, Unit testing for domain-specific languages, in: W. M. Taha (Ed.), Domain-Specific Languages, Springer, 2009, pp. 125–147.

[6] B. Meyers, J. Denil, I. Dávid, H. Vangheluwe, Automated testing support for reactive domain-specific modelling languages, in: Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, ACM, 2016, pp. 181–194.

[7] F. Khorram, E. Bousse, J.-M. Mottu, G. Sunyé, Adapting tdl to provide testing support for executable dsls, Journal of Object Technology 20 (3) (2021) 6:1–15.

[8] P. C. Cañizares, P. Gómez-Abajo, A. Núñez, E. Guerra, J. de Lara, 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, 2021, pp. 49–54.

[9] F. Khorram, E. Bousse, J.-M. Mottu, G. Sunyé, Advanced Testing and Debugging Support for Reactive Executable DSLs, Software and Systems Modeling (2022).

[10] P. Ammann, J. Offutt, Introduction to software testing, Cambridge University Press, 2016.

[11] W. E. Wong, R. Gao, Y. Li, R. Abreu, F. Wotawa, A survey on software fault localization, IEEE Transactions on Software Engineering 42 (8) (2016) 707–740.

[12] J. Troya, S. Segura, J. A. Parejo, A. Ruiz-Cortés, Spectrum-based fault localization in model transformations, ACM Transactions on Software Engineering and Methodology (TOSEM) 27 (3) (2018) 1–50.

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

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

[15] D. Steinberg, F. Budinsky, E. Merks, M. Paternostro, EMF: eclipse modeling framework, Addison-Wesley, 2008.

[16] E. Bousse, T. Mayerhofer, B. Combemale, B. Baudry, A generative approach to define rich domain-specific trace metamodels, in: G. Taentzer, F. Bordeleau (Eds.), Modelling Foundations and Applications, Springer, 2015, pp. 45–61.

[17] E. Bousse, T. Mayerhofer, B. Combemale, B. Baudry, Advanced and efficient execution trace management for executable domain-specific modeling languages, Software and Systems Modeling (2019) 1–37doi:10.1007/s10270-017-0598-5.
URL https://hal.inria.fr/hal-01614377

[18] A. Iung, J. Carbonell, L. Marchezan, E. Rodrigues, M. Bernardino, F. P. Basso, B. Medeiros, Systematic mapping study on domain-specific language development tools, Empirical Software Engineering 25 (5) (2020) 4205–4249. doi:10.1007/s10664-020-09872-1.
URL https://doi.org/10.1007/s10664-020-09872-1

[19] S. Erdweg, T. van der Storm, M. Völter, L. Tratt, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, G. Konat, P. J. Molina, M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. Vergu, E. Visser, K. van der Vlist, G. Wachsmuth, J. van der Woning, Evaluating and comparing language workbenches: Existing results and benchmarks for the future, Computer Languages, Systems & Structures 44 (2015) 24–47. doi:10.1016/j.cl.2015.08.007.
URL http://www.sciencedirect.com/science/article/pii/S1477842415000573

[20] H. A. de Souza, M. L. Chaim, F. Kon, Spectrum-based software fault localization: A survey of techniques, advances, and challenges, arXiv preprint arXiv:1607.04347 (2016).

[21] M. L. Mohd-Shafie, W. M. N. W. Kadir, H. Lichter, M. Khatibsyarbini, M. A. Isa, Model-based test case generation and prioritization: a systematic literature review, Software and Systems Modeling (2021) 1–37doi:10.1007/s10270-021-00924-8.
URL https://doi.org/10.1007/s10270-021-00924-8

[22] R. Bendraou, B. Combemale, X. Crégut, M.-P. Gervais, Definition of an eXecutable SPEM 2.0, in: Proceedings of the 14th Asia-Pacific Software Engineering Conference (APSEC), IEEE Computer Society, 2007, pp. 390–397.

[23] OASIS, Web services business process execution language version 2.0 (2007).
URL http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html

[24] O. M. G. (OMG), Semantics of a foundational subset for executable uml models, https://www.omg.org/spec/FUML/, (last accessed in September 2022) (2022).

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

[26] E. Biermann, C. Ermel, G. Taentzer, 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, 2008, p. 53–67.

[27] B. Danglot, O. Vera-Perez, Z. Yu, A. Zaidman, M. Monperrus, B. Baudry, A snowballing literature study on test amplification, Journal of Systems and Software 157 (2019) 110398.

[28] E. Bousse, D. Leroy, B. Combemale, M. Wimmer, B. Baudry, Omniscient Debugging for Executable DSLs, Journal of Systems and Software 137 (2018) 261–288. doi:10.1016/j.jss.2017.11.025.
URL https://hal.inria.fr/hal-01662336

[29] J. A. Jones, M. J. Harrold, 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, 2005, p. 273–282. doi:10.1145/1101908.1101949.
URL https://doi.org/10.1145/1101908.1101949

[30] L. Naish, H. J. Lee, K. Ramamohanarao, A model for spectra-based software diagnosis, ACM Transactions on software engineering and methodology (TOSEM) 20 (3) (2011) 1–32.

[31] F. Khorram, A. Garmendia, A Language-Parametric Test Coverage Framework for DSLs: Artefacts (2023). doi:10.5281/zenodo.7709671.
URL https://doi.org/10.5281/zenodo.7709671

[32] Object Management Group, Precise semantics of uml state machines (2019).
URL https://www.omg.org/spec/PSSM/1.0/

[33] J. Cangussu, J. Palsberg, V. Samanta, Modern Compiler Implementation in Java: the MiniJava Project, https://www.cambridge.org/resources/052182060X, [Online; accessed 28-February-2023] (2023).

[34] E. team, Jacoco java code coverage library, [Online; accessed 28-February-2023] (2023).
URL https://github.com/jacoco/jacoco

[35] CodeCover, Codecover glass-box testing tool, [Online; accessed 28-February-2023] (2023).
URL http://codecover.org

[36] OpenClover, Openclover code coverage tool for java and groovy, [Online; accessed 28-February-2023] (2023).
URL https://openclover.org/

[37] E. Mandrikov, What kind of branching coverage is measured by jacoco, [Online; accessed 28-February-2023] (2023).
URL https://groups.google.com/g/jacoco/c/b8bAWaWPl6I/m/eMKixUpMCAAJ

[38] P. Gómez-Abajo, E. Guerra, J. de Lara, M. G. Merayo, Wodel-test: a model-based framework for language-independent mutation testing, Software and Systems Modeling 20 (2020) 1–27.

[39] E. Compare, EMF Compare, https://www.eclipse.org/emf/compare, [Online; accessed 28-February-2023] (2023).

[40] X. Xie, T. Y. Chen, F.-C. Kuo, B. Xu, A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization, ACM Trans. Softw. Eng. Methodol. 22 (4) (oct 2013). doi:10.1145/2522920.2522924.
URL https://doi.org/10.1145/2522920.2522924

[41] CodeCover, Codecover measurement under java: Statement coverage, [Online; accessed 28-February-2023] (2023).
URL http://codecover.org/documentation/references/javaMeasurement.html#StatementCovereage

[42] CodeCover, Codecover measurement under java: Branch coverage, [Online; accessed 28-February-2023] (2023).
URL http://codecover.org/documentation/references/javaMeasurement.html#BranchCovereage

[43] D. Zou, J. Liang, Y. Xiong, M. D. Ernst, L. Zhang, An empirical study of fault localization families and their combinations, IEEE Trans. Software Eng. 47 (2) (2021) 332–347. doi:10.1109/TSE.2019.2892102.
URL https://doi.org/10.1109/TSE.2019.2892102

[44] X. Mao, Y. Lei, Z. Dai, Y. Qi, C. Wang, Slice-based statistical fault localization, Journal of Systems and Software 89 (2014) 51–62. doi:10.1016/j.jss.2013.08.031.
URL https://www.sciencedirect.com/science/article/pii/S0164121213002185

[45] R. Abreu, P. Zoeteweij, A. J. van Gemund, Spectrum-based multiple fault localization, in: 2009 IEEE/ACM International Conference on Automated Software Engineering, 2009, pp. 88–99. doi:10.1109/ASE.2009.25.

[46] A. Maxwell, A. Pilliner, Deriving coefficients of reliability and agreement for ratings, British Journal of Mathematical and Statistical Psychology 21 (1) (1968) 105–116. doi:10.1111/j.2044-8317.1968.tb00401.x.

[47] L. Naish, H. J. Lee, K. Ramamohanarao, A model for spectra-based software diagnosis, ACM Trans. Softw. Eng. Methodol. 20 (3) (aug 2011). doi:10.1145/2000791.2000795.
URL https://doi.org/10.1145/2000791.2000795

[48] W. E. Wong, V. Debroy, R. Gao, Y. Li, The dstar method for effective software fault localization, IEEE Transactions on Reliability 63 (1) (2014) 290–308. doi:10.1109/TR.2013.2285319.

[49] Y. Qi, X. Mao, Y. Lei, C. Wang, Using automated program repair for eval-

uating the effectiveness of fault localization techniques, in: Proceedings of the 2013 International Symposium on Software Testing and Analysis, ISSTA 2013, Association for Computing Machinery, New York, NY, USA, 2013, p. 191–201. doi:10.1145/2483760.2483785.
URL https://doi.org/10.1145/2483760.2483785

[50] W. E. Wong, V. Debroy, Y. Li, R. Gao, Software fault localization using dstar (d*), in: 2012 IEEE Sixth International Conference on Software Security and Reliability, 2012, pp. 21–30. doi:10.1109/SERE.2012.12.

[51] F. Y. Assiri, J. M. Bieman, Fault localization for automated program repair: effectiveness, performance, repair correctness, Software Quality Journal 25 (1) (2017) 171–199. doi:doi:10.1007/s11219-016-9312-z.
URL https://doi.org/10.1007/s11219-016-9312-z

[52] R. Abreu, P. Zoeteweij, R. Golsteijn, A. J. van Gemund, A practical evaluation of spectrum-based fault localization, Journal of Systems and Software 82 (11) (2009) 1780–1792, sI: TAIC PART 2007 and MUTATION 2007. doi:10.1016/j.jss.2009.06.035.
URL https://www.sciencedirect.com/science/article/pii/S0164121209001319

[53] T. Janssen, R. Abreu, A. J. van Gemund, Zoltar: A spectrum-based fault localization tool, in: Proceedings of the 2009 ESEC/FSE Workshop on Software Integration and Evolution @ Runtime, SINTER '09, Association for Computing Machinery, New York, NY, USA, 2009, p. 23–30. doi:10.1145/1596495.1596502.
URL https://doi.org/10.1145/1596495.1596502

[54] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, A. Wesslén, Experimentation in software engineering, Springer Science & Business Media, 2012.

[55] Cobertura, A code coverage utility for java, [Online; accessed 28-February-2023] (2023).
URL http://cobertura.github.io/cobertura/

[56] J. Chen, Y. Bai, D. Hao, Y. Xiong, H. Zhang, B. Xie, Learning to prioritize test programs for compiler testing, in: S. Uchitel, A. Orso, M. P. Robillard (Eds.), Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017, IEEE / ACM, 2017, pp. 700–711. doi:10.1109/ICSE.2017.70.
URL https://doi.org/10.1109/ICSE.2017.70

[57] J. Chen, J. Patra, M. Pradel, Y. Xiong, H. Zhang, D. Hao, L. Zhang, A survey of compiler testing, ACM Comput. Surv. 53 (1) (2021) 4:1–4:36. doi:10.1145/3363562.
URL https://doi.org/10.1145/3363562

[58] W. Cazzola, L. Favalli, The language mutation problem: Leveraging language product lines for mutation testing of interpreters, J. Syst. Softw. 195 (2023) 111533. doi:10.1016/J.JSS.2022.111533.
URL https://doi.org/10.1016/j.jss.2022.111533

[59] M. El qortobi, A. Rahj, J. Bentahar, R. Dssouli, Test generation tool for modified condition/decision coverage: Model based testing, in: Proceedings of the 13th International Conference on Intelligent Systems: Theories and Applications, 2020, pp. 1–6.

[60] T. Waheed, M. Z. Z. Iqbal, Z. I. Malik, Data flow analysis of uml action semantics for executable models, in: European Conference on Model Driven Architecture-Foundations and Applications, Springer, 2008, pp. 79–93.

[61] P. N. Boghdady, N. L. Badr, M. A. Hashim, M. F. Tolba, An enhanced test case generation technique based on activity diagrams, in: Proceedings of the International Conference on Computer Engineering & Systems, IEEE, 2011, pp. 289–294.

[62] I. Baxter, Branch coverage for arbitrary languages made easy: Transformation systems to the rescue, IW APA TV2/IC SE2001. http://techwell.com/sites/default/files/articles/XUS1173972file1_0. pdf (2001).

[63] S. Yue, J. Gray, SPOT: A DSL for extending fortran programs with metaprogramming, Adv. Softw. Eng. 2014 (2014) 917327:1–917327:23.

[64] J. Misurda, J. A. Clause, J. L. Reed, B. R. Childers, M. L. Soffa, Demand-driven structural testing with dynamic instrumentation, in: Proceedings of the 27th International Conference on Software Engineering (ICSE), IEEE, 2005, pp. 156–165.

[65] M. Bordin, C. Comar, T. Gingold, J. Guitton, O. Hainque, T. Quinot, J. Delange, J. Hugues, L. Pautet, Couverture: an innovative open framework for coverage analysis of safety critical applications, Ada User Journal 30 (4) (2009) 248–255.

[66] K. Sakamoto, K. Shimojo, R. Takasawa, H. Washizaki, Y. Fukazawa, 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, 2013, pp. 422–430.

[67] OpenClover, Using clover for other programming languages, [Online; accessed 28-February-2023] (2023).
URL https://openclover.org/doc/manual/latest/hacking--using-openclover-for-other-programming-languages.html

[68] CodeCover, Add a new programming language, [Online; accessed 28-February-2023] (2023).
URL http://codecover.org/development/programming-language.html

[69] JavaCC, Java compiler compiler, [Online; accessed 28-February-2023] (2023).
URL https://javacc.github.io/javacc/

[70] P. Li, M. Jiang, Z. Ding, Fault localization with weighted test model in model transformations, IEEE Access 8 (2020) 14054–14064.

[71] M. Raselimo, B. Fischer, Spectrum-based fault localization for context-free grammars, in: Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering, 2019, pp. 15–28.

[72] K. Wang, A. Sullivan, D. Marinov, S. Khurshid, Fault localization for declarative models in alloy, in: Proceedings of the 31st International Symposium on Software Reliability Engineering (ISSRE), IEEE, 2020, pp. 391–402.

[73] L. Arcega, J. Font, Ø. Haugen, C. Cetina, An approach for bug localization in models using two levels: model and metamodel, Software and Systems Modeling 18 (6) (2019) 3551–3576.

[74] L. Arcega, J. Font, C. Cetina, 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, 2018, pp. 90–100.

[75] L. Arcega, J. F. Arcega, Ø. Haugen, C. Cetina, Bug localization in model-based systems in the wild, ACM Transactions on Software Engineering and Methodology (TOSEM) 31 (1) (2021) 1–32.

[76] R. Casamayor, L. Arcega, F. Pérez, C. Cetina, Bug localization in game software engineering: evolving simulations to locate bugs in software models of video games, in: E. Syriani, H. A. Sahraoui, N. Bencomo, M. Wimmer (Eds.), Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems, MODELS 2022, Montreal, Quebec, Canada, October 23-28, 2022, ACM, 2022, pp. 356–366.

[77] K-Framework, K semantic framework, [Online; accessed 18-October-2023] (2023).
URL https://kframework.org/

Appendix A. SBFL Techniques

We implemented in our framework 18 existing formulas that are listed in Table A.7. These formulas were reused from the work of Troya et al. [12] who investigated a large set of primary studies proposing concrete SBFL techniques, and applied them to locate faulty rules in model transformations.

Table A.7: SBFL techniques adapted for model element suspiciousness measurement

Technique	Formula
Arithmetic Mean [40]	$\frac{2(N_{CF} \times N_{US} - N_{UF} \times N_{CS})}{(N_{CF} + N_{CS}) \times (N_{US} + N_{UF}) + (N_{CF} + N_{UF}) \times (N_{CS} + N_{US})}$
Barinel [45]	$1 - \frac{N_{CS}}{N_{CS} + N_{CF}}$
Baroni-Urbani & Buser [50]	$\frac{\sqrt{N_{CF} \times N_{US}} + N_{CF}}{\sqrt{N_{CF} \times N_{US}} + N_{CF} + N_{CS} + N_{UF}}$
Braun-Banquet [11]	$\frac{N_{CF}}{max(N_{CF} + N_{CS}, N_{CF} + N_{UF})}$
Cohen [47]	$\frac{2 \times (N_{CF} \times N_{US} - N_{UF} \times N_{CS})}{(N_{CF} + N_{CS}) \times (N_{US} + N_{CS}) + (N_{CF} + N_{UF}) \times (N_{UF} + N_{US})}$
DStar [48]	$\frac{(N_{CF})^*}{N_{CS} + N_F + N_{CF}}$
Kulczynski2 [47]	$\frac{1}{2} \times (\frac{N_{CF}}{N_{CF} + N_{UF}} + \frac{N_{CF}}{N_{CF} + N_{CS}})$
Mountford [50]	$\frac{N_{CF}}{0.5 \times ((N_{CF} \times N_{CS}) + (N_{CF} \times N_{UF})) + (N_{CS} \times N_{UF})}$
Ochiai [52]	$\frac{N_{CF}}{\sqrt{N_F \times (N_{CF} + N_{CS})}}$
Ochiai2 [51]	$\frac{N_{CF} \times N_{US}}{\sqrt{(N_{CF} + N_{CS}) \times (N_{US} + N_{UF}) \times (N_{CF} + N_{UF}) \times (N_{CS} + N_{US})}}$
Op2 [47]	$N_{CF} - \frac{N_{CS}}{N_S + 1}$
Phi [46]	$\frac{N_{CF} \times N_{US} - N_{UF} \times N_{CS}}{\sqrt{(N_{CF} + N_{CS}) \times (N_{CF} + N_{UF}) \times (N_{CS} + N_{US}) \times (N_{UF} + N_{US})}}$
Pierce [11]	$\frac{(N_{CF} \times N_{UF}) + (N_{UF} \times N_{CS})}{(N_{CF} \times N_{UF}) + (2 \times N_{UF} \times N_{US}) + (N_{CS} \times N_{US})}$
Rogers & Tanimoto [44]	$\frac{N_{CF} + N_{US}}{N_{CF} + N_{US} + 2(N_{UF} + N_{CS})}$
Russel-Rao [49]	$\frac{N_{CF}}{N_{CF} + N_{UF} + N_{CS} + N_{US}}$
Simple Matching [11]	$\frac{N_{CF} + N_{US}}{N_{CF} + N_{CS} + N_{US} + N_{UF}}$
Tarantula [29]	$\frac{\frac{N_{CF}}{N_F}}{\frac{N_{CF}}{N_F} + \frac{N_{CS}}{N_S}}$
Zoltar [53]	$\frac{N_{CF}}{N_{CF} + N_{UF} + N_{CS} + \frac{10000 \times N_{UF} \times N_{CS}}{N_{CF}}}$