# Adapting TDL to Provide Testing Support for Executable DSLs

**Faezeh Khorram**[*], **Erwan Bousse**[†], **Jean-Marie Mottu**[†*], **and Gerson Sunyé**[†]
[*]IMT Atlantique, LS2N, Nantes, France
[†]Université de Nantes, LS2N, Nantes, France

**ABSTRACT** Testing is one of the most prevalent and successful verification and validation (V&V) techniques used in the field of software engineering. While a large number of testing frameworks exist for general-purpose programming languages, providing testing facilities for any given executable Domain Specific Language (xDSL) remains a costly and challenging task. In this context, a standard such as the Test Description Language (TDL) appears as a suitable foundation for the definition of a generic testing approach for xDSLs. Unfortunately, TDL does not provide the domain-specific concepts required to write test cases for a given xDSL and does not include any model execution facilities. Our contribution addresses these limitations and thereby provides a fully generic testing approach for xDSLs based on TDL. Required TDL domain-specific concepts are automatically inferred from the xDSL definition through a model transformation into TDL. Model execution facilities are provided through the definition of a refined operational semantics for TDL. The application of our approach on 5 different xDSLs shows its generality and that it can successfully be used for testing executable models.

**KEYWORDS** Executable Domain-Specific Language, Executable models, Testing, Test Description Language

## 1. Introduction

A large portion of DSLs are proposed for describing the dynamic aspects of systems as behavioral models (e. g., (Object Management Group 2013b; Bendraou et al. 2007; OASIS 2007; Fischer et al. 2000)). Each time a new DSL is engineered, a complete modeling environment has to be provided for its users (i. e., domain experts), so they can use the DSL in practice. When the environment offers dynamic verification and validation (V&V) techniques, the domain expert can also analyze the behavioral models as early as possible to ensure the correctness of the system's behavior. As dynamic V&V techniques rely on the ability to execute models, their application is reserved to DSLs with execution semantics, such as DSLs with translational semantics (i. e., compilation) or operational semantics (i. e., interpretation). In this paper, we focus on DSLs with operational semantics, referred to as *executable DSLs (xDSLs)*.

**JOT reference format:**
Faezeh Khorram, Erwan Bousse, Jean-Marie Mottu, and Gerson Sunyé. *Adapting TDL to Provide Testing Support for Executable DSLs.* Journal of Object Technology. Vol. vv, No. nn, yyyy. Licensed under Attribution - NonCommercial - No Derivatives 4.0 International (CC BY-NC-ND 4.0)
http://dx.doi.org/10.5381/jot.yyyy.vv.nn.aa

In the field of software engineering, probably the most prevalent dynamic V&V technique is *testing*, which involves executing systems and observing whether they act as expected. Accordingly, testing frameworks have been built for both a wide range of General-Purpose Languages (GPLs) (e. g. JUnit for Java) and specific xDSLs (Mijatov et al. 2015; Kos et al. 2016; Lübke & van Lessen 2017; Iqbal et al. 2019a). A testing framework must at least include both a way to write test cases, and a way to execute such test cases in unison with the programs or models under test. Unfortunately, providing such complex testing facilities for a given *new* xDSL remains an expensive and error-prone task.

Therefore, in a context where the engineering of new xDSLs is recurrent, a desirable solution would be an approach applicable to a wide range of xDSLs, i. e. a *generic* testing approach for xDSLs. This raises at least three interconnected challenges. First, to allow the domain expert to write test cases, a testing language must be defined, generated, or identified. In particular, this testing language must somehow allow the domain expert to use domain concepts to define how a model under test should be executed, and what results should be expected from the execution. Second, the execution semantics of this testing language must somehow be connected to the execution semantics of the considered xDSL, for the testing language to demand the execution of models as needed. Third, this testing language must provide facilities to analyze the runtime state of the tested model, and to compare this state with the expected one.

A recent effort of the European Telecommunications Standards Institute (ETSI) led to the creation of the Test Description Language (TDL), a standardized language for the specification of test descriptions (Makedonski et al. 2019). Since TDL is not specific to any specific GPL or xDSL, it represents an interesting candidate for generically writing test cases for executable models. In addition, TDL was designed as a simple language for testers lacking programming knowledge, making it a good fit for domain experts working on models. Unfortunately, TDL fails to fully address the three aforementioned challenges: (1) because of its genericity, TDL requires the domain expert to first define the required domain-specific concepts, before being able to write test cases; (2) the TDL standard does not provide any clear way to make TDL test cases able to execute models conforming to a given xDSL; (3) the TDL standard relies on a simple representation of the expected observable behavior of the system under test, and does not provide any efficient way to analyze an arbitrarily complex runtime state of a tested model.

This paper addresses these limitations and thereby proposes *a novel generic testing approach for xDSLs*. This approach uses TDL as a testing language and relies on three main contributions. First, we provide a model transformation to automatically generate a TDL library—i. e. all the TDL boilerplate code that the domain expert would otherwise write by hand—from the definition of an xDSL. Such generated TDL library can be used by the domain expert to write test cases for models conforming to the considered xDSL. This transformation relies on a mapping between the concepts of the Ecore metamodeling language (Steinberg et al. 2008) and the concepts of TDL. Second, we provide an operational semantics for TDL, adapted to the testing of executable models. To be compatible with a wide range of diverse xDSLs, this operational semantics for TDL is not coupled to any specific xDSL, nor to any specific metaprogramming approach used to define the considered xDSL. Third, the approach provides two different methods to interrogate the state of the runtime model: one relying on model comparison and the second relying on an OCL interpreter. This enables the definition of oracles for executable models in TDL test cases.

We implemented the presented approach for the GEMOC Studio, a language and modeling workbench for xDSLs (Bousse et al. 2016). We conducted an evaluation to assess the *genericity* aspect of our proposed approach considering the diversity of xDSLs. More precisely, we aimed to answer the following research questions: **RQ#1** Does the approach provide testing facilities for xDSLs in which their abstract syntax is designed for *different domains*? **RQ#2** Does the approach provide testing facilities for xDSLs in which their operational semantics is implemented using *different metaprogramming approaches*? To this end, we applied the approach on five xDSLs covering various domains and implemented with different metaprogramming approaches. The evaluation results demonstrate that the genericity aspect is successfully realized.

The rest of the paper is organized as follows: Section 2 provides the background and presents a running example for the paper. Section 3 describes an overview of our proposed approach, and the details for its main components are given in Sections 4 and 5. The tool support for the approach is described in Section 6. In Section 7, the method used for the evaluation along with its result is illustrated. The previous related work is presented in Section 8 and the paper concludes in 9 with a discussion on future work.

## 2. Background and Motivation

In this section, we first present the executable DSLs considered in the scope of the approach and give an overview of the Test Description Language (TDL). Then, we further motivate the proposed approach through a running example.

### 2.1. Executable DSLs (xDSLs)

In the present paper, we focus on executable DSLs (xDSLs), which are composed of two main parts: an abstract syntax[1] and an operational semantics. The abstract syntax defines the domain concepts while the operational semantics (i. e. the interpreter) specifies how the runtime state of a conforming model under execution varies over time. More specifically, we consider the abstract syntax of an xDSL to be defined as a metamodel using the Ecore metamodeling language (Steinberg et al. 2008). A metamodel is commonly made of a set of metaclasses, each containing a set of features. A feature can be either an attribute typed by a primitive type or a reference to another metaclass.

The operational semantics of an xDSL is composed of two parts: the definition of what are the possible runtime states of a model under execution and the set of execution rules that define how such a runtime state changes over time. For a given xDSL, we consider that the definition of the possible runtime states is a set of *dynamic features* directly added to the metamodel. During the execution of a model, only the parts defined by these dynamic features may be altered by the execution rules. To distinguish the dynamic features from other ones, we assume they each have an annotation labeled ‘dynamic’.

Figure 1 illustrates an example of an executable Finite State Machine DSL (later referred to as xFSM). An xFSM model is an automaton that consumes a string and produces a new string based on the transitions that were fired to parse the consumed string. The abstract syntax of xFSM is defined as a metamodel (part **a** in Figure 1). The root element of an xFSM model is a StateMachine composed of a set of State and Transition elements. The initialState of the StateMachine specifies the starting state. Each Transition object connects two states called source and target. A transition may also have an input string that tells which characters are consumed when this transition is fired, and an output string that tells what characters are produced when this transition is fired. The elements written in bold are

[1] The proposed approach is agnostic of the *concrete syntax* of an xDSL, therefore this part is left out of the scope of the paper.

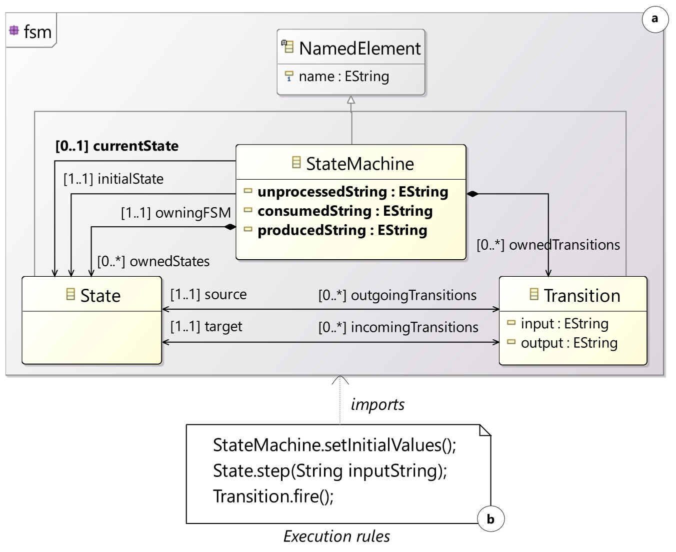


**Figure 1** Executable Finite State Machine DSL (xFSM)

the dynamic features that define the possible runtime states of an xFSM model.

We consider that the operational semantics of xFSM comprises three execution rules (part **b** in Figure 1). These execution rules can be written using any language or framework able to perform in-line model transformations. For example, Listing 1 shows the xFSM execution rules implemented using the Action Language for EMF (ALE) (Leduc et al. 2017). ALE provides three annotations to decorate methods: *@init* to designate the method that must be called to initialize the dynamic state of the model before its execution, *@main* to designate the method that must be called to start the execution of the model, and *@step* to designate operations that perform observable execution steps. The `setInitialValues` rule sets the initial values of the dynamic features of the model, namely the currentState (set to the initialState), the unprocessedString (set to the input string given to the model), the consumedString and the producedString (both set to empty strings) (lines 4–7). The model execution starts wtih the `main()` method (lines 9–14) which calls the `step(String inputString)` rule on the currentState with the unprocessedString as its parameter (line 11) while the unprocessedString is not empty (line 10). The `step(String inputString)` rule checks the outgoingTransitions of the called State to find the one whose input is equal to the first character of the rule parameter (i. e., the inputString), and then calls the `fire` rule on that valid transition (lines 18–23). The `fire` rule is responsible of changing the model state (i. e., the bold elements in part **a** of Figure 1). To do so, it sets the currentState to the target of the fired transition, removes the first character of the unprocessedString, and concatenates the input and the output of the fired transition to the end of the consumedString and the producedString, respectively (lines 27–40).

Figure 3 shows an example of a model conforming to xFSM. It describes a StateMachine designed to perform a *bit shifting* operation on a sequence of bits (i. e., the unprocessedString). The execution of the model starts from `S0` (i. e., the initialState) and stops when the bit sequence is shifted. Note that there is

```
 1  behavior fsm;
 2  open class StateMachine {
 3    @init
 4    def void setInitialValues (String input) {
 5      self.currentState := self.initialState;
 6      self.unprocessedString := input;
 7    }
 8    @main
 9    def void main() {
10      while (self.unprocessedString.size() > 0) {
11        self.currentState.step(
12          self.unprocessedString);
13      }
14    }
15  }
16  open class State {
17    @step
18    def void step(String inputString) {
19      Sequence(fsm::Transition) validTrans:=
20        self.outgoingTransitions->select(
21          t | inputString.startsWith(t.input));
22      validTrans->at(1).fire();
23    }
24  }
25  open class Transition {
26    @step
27    def void fire() {
28      StateMachine fsm := self.source.owningFSM;
29      fsm.currentState := self.target;
30      fsm.producedString += self.output;
31      fsm.consummedString += self.input;
32      if(self.input.size()+1 <=
33        fsm.unprocessedString.size()) {
34        fsm.unprocessedString :=
35          fsm.unprocessedString
36          .substring(self.input.size()+1);
37      }else {
38        fsm.unprocessedString := '';
39      }
40    }
41  }
```

**Listing 1** The operational semantics of the xFSM DSL implemented using the ALE language

a defect in this model because the state `S1` cannot process an input bit with a value of 1. A possible fix would be a cyclic transition for `S1` whose `input` and `output` properties are equal to 1. We aim to be able to detect this defect with a test suite written and executed using the approach proposed in this paper.

## 2.2. The Test Description Language (TDL)

The Test Description Language (TDL) is a DSL for describing test cases introduced by the European Telecommunications Standards Institute (ETSI). The main objective of TDL is to provide a common perception of test cases for different stakeholders, through filling the gap between the abstract test requirements described by non-technical people and the complex code of executable test cases. TDL supports the high-level description of test objectives derived from system requirements, the definition of test cases refining these objectives, and also the presentation of the test execution results (Makedonski et al. 2019). The TDL standard includes both, a loose semantics written in natural language and a precise translational semantics using the Test-

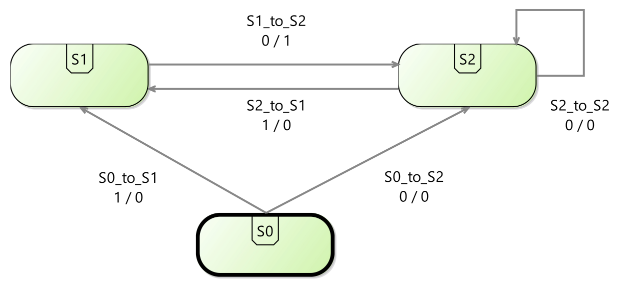


**Figure 3** A sample xFSM model for performing bit shifting operation on a sequence of bits. It has a defect since S1 cannot process an input bit with a value of 1

ing and Test Control Notation version 3 (TTCN-3) as a target language—TTCN-3 being also standardized by the ETSI. The ETSI also provides a reference implementation of TDL built atop the Eclipse Modeling Framework (EMF) and available as an open-source project[2]. Its current implementation includes a standard abstract syntax, textual and graphical concrete syntaxes, as well as tools for model validation, transformation to TTCN-3, translation to XMI format, and automatic documentation generation in Word.

Listing 2 shows an example of a TDL model for testing the bit shifting FSM shown in Figure 3. It contains all the essential elements for writing a simple test case; they conform to the TDL metamodel briefly presented in Figure 2. A Package is the root element of a TDL model, and thus is the container of all other elements. Using various DataType elements, a tester can define the data types required for the definition of test data that will be exchanged between the test suite (later referred to as the *test system*) and the System Under Test (SUT). For instance for testing the bit shifting FSM, the tester defines a data type named `EString` (line 2) and another named `StateMachine` (lines 3–7). `StateMachine` is a StructuredDataType containing Members of different types, such as `unprocessedString` and `producedString` of type `EString`.

A TestConfiguration element specifies how the test system will communicate with the SUT. It contains at least one Tester and exactly one SUT component instances, with connections between their gates. Each component instance has a ComponentType, which specifies the component communication channels using the so-called gates. Accordingly, a ComponentType contains at least one gate that is instantiated from a GateType. A Gate Type specifies the kinds of data that can be exchanged through the gates instantiated from it. For example in Listing 2, the `FSMGate` Gate Type accepts data of type `StateMachine` (line 9). There is one ComponentType named `FSMComponent` (lines 10–12), that has one gate instance named g of type `FSMGate`. The Test Configuration named `FSMConfiguration` (lines 13–17) has one Tester component instance (`FSMTester` in line 14), one SUT (`FSM_SUT` in line 15), and connections between their gates (line 16). This configuration means the `FSMTester` and the `FSM_SUT` can exchange data of type `StateMachine` through their g gates.

Test data can be defined through instantiation of DataType elements. This concerns both the input data that will be sent to the SUT during test case execution, and the expected output data that will be used in assertions (i. e. to define the oracle of the test case). In the TDL model shown in Listing 2, the only test data is `bitShifting` (line 19).

A test case is defined using a TestDescription element, which uses one of the previously defined instances of TestConfiguration, and contains a sequence of Behavior elements. At the moment, there are twenty types of Behavior elements in TDL, such as Message, TimeOut, AlternativeBehavior, BoundedLoopBehavior, etc. For example in Listing 2, the `bitShiftingTest` Test Description uses

[2] https://labs.etsi.org/rep/top/ide

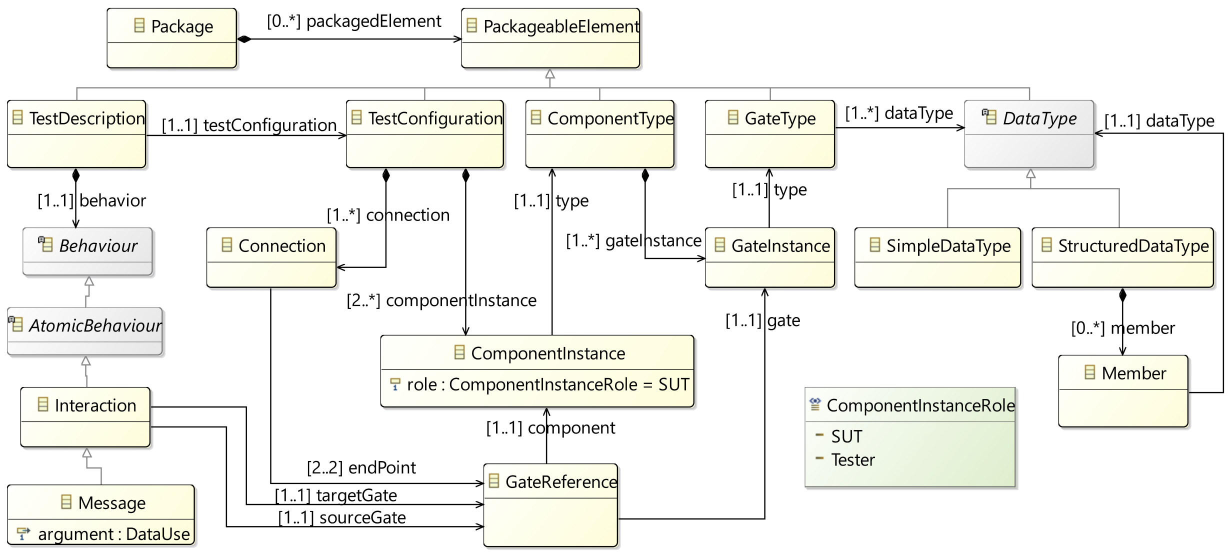


**Figure 2** An Excerpt of the TDL Metamodel (ETSI ES 203 119-1 2020)

```
 1 Package FSMTest{
 2   Type EString;
 3   Type StateMachine(
 4     _name of type EString,
 5     unprocessedString of type EString,
 6     producedString of type EString
 7   );
 8
 9   Gate Type FSMGate accepts StateMachine;
10   Component Type FSMComponent having {
11     gate g of type FSMGate;
12   }
13   Test Configuration FSMConfiguration {
14     create Tester FSMTester of type FSMComponent;
15     create SUT FSM_SUT of type FSMComponent;
16     connect FSMTester.g to FSM_SUT.g;
17   }
18
19   StateMachine bitShifting (_name="BitShifting");
20
21   // Start of the actual test case
22   Test Description bitShiftingTest uses
23       configuration FSMConfiguration {
24
25     // Sending test input data
26     FSMTester.g sends bitShifting
27     (unprocessedString="10010110") to FSM_SUT.g;
28     // Oracle with expected output data
29     FSM_SUT.g sends bitShifting
30     (producedString="01001011") to FSMTester.g;
31   }
32 }
```

**Listing 2** An example of a TDL model with a test case for the bit shifting FSM

the `FSMConfiguration` and contains two Message elements (lines 22–31). A Message specifies a sending of data through the connected gates of the tester and SUT components.When the sender of a Message is the Tester, the sent data corresponds to test input data. But when the SUT is the sender, the Message is actually an assertion that the SUT *should* send back said Message—and the data it contains. In Listing 2, the first message specifies the test input data—which is an FSM in a runtime state where its `unprocessedString` is set as "10010110" (lines 26–27)—and the second message asserts that the `producedString` of the FSM is equal to "01001011" (lines 29–30). Note that, in theory, if we were to execute this test case, it would not pass since the bitShifting FSM (i. e., the model under test) cannot process two consecutive 1 bits. This test case should therefore be able to detect this defect.

### 2.3. Limitations of TDL and Motivation

The presented example shows that TDL can be used to successfully describe test cases for an executable model (xModel). However, producing a test case for an xModel such as the one presented in Listing 2, the domain expert faces the three limitations of TDL already identified in the introduction.

First, all the required data types (e. g. lines 2–7) and test configurations (e. g. lines 13–17) have to be manually defined by the domain expert using the xDSL definition (i. e. the abstract syntax and the operational semantics). Yet, not only the domain expert is unlikely to be knowledgeable of the internals of the xDSL, but this endeavor is also costly and error-prone.

Second, we have no actual way to execute the test case from Listing 2. While a translational semantics using TTCN-3 as a target language exists for TDL, this semantics is only partial, and mainly aims to manage test cases for software systems communicating through common protocols (TCP, UDP, TELNET, SQL, HTTP, etc.). Therefore, a new execution semantics for TDL adapted to the testing of executable models is required to execute the TDL test case shown in Listing 2.

Third, the expressiveness of TDL is rather limited when it comes to analyzing the content of the runtime state of the tested model and thus when it comes to writing assertions. Since tested models may be large, their runtime states may be numerous and complex. Therefore, a proper way to query their contents from a TDL test case appears necessary.

In the next sections, we present our approach to provide testing facilities to xDSLs using TDL. We address all aforementioned limitations with, respectively, a generator to automatically produce the required TDL data types and test configurations, an operational semantics for TDL adapted to the testing of executable models, and a way to use queries written using the Object Constraint Language (OCL) in TDL test cases; so the query validation result can be used as the actual output to be asserted with the expected output specified in the test case.

## 3. Approach Overview

Figure 4 presents an overview of the proposed approach. At the top left corner, we assume that an xDSL was implemented by a language engineer according to the definitions given in Section 2.1. At the center, the domain expert uses the provided xDSL to define an executable model and wishes to write TDL test cases for this model.

At the top is shown the first component of the approach, namely the *TDL Library Generator*. Its purpose is to automatically generate a domain-specific TDL Library that the domain expert can then use to conveniently write test cases for executable models conforming to the xDSL. This library provides all the data types required for the specification of test data, a set of default test configurations, some elements for enabling TDL test cases to request the execution of the model under test, and some elements for writing OCL queries in the TDL test cases. As shown in Figure 4, the library generator requires as input data the definition of the xDSL. In particular, the abstract syntax and the part of the operational semantics defining the possible runtime states of the conforming models.

At the bottom-right corner is shown the second component of the approach, namely the *TDL Interpreter*. This interpreter is based on an operational semantics for TDL, adapted to the testing of executable models. For this purpose, this operational semantics is connected to two external components: the *Execution Engine* and the *Query Evaluator*. We assume that the *Execution Engine* exists and provides services to trigger the execution of a model conforming to an xDSL. In particular, that this engine is able to load a model, load an xDSL, and execute the model using the operational semantics of the xDSL. Here,

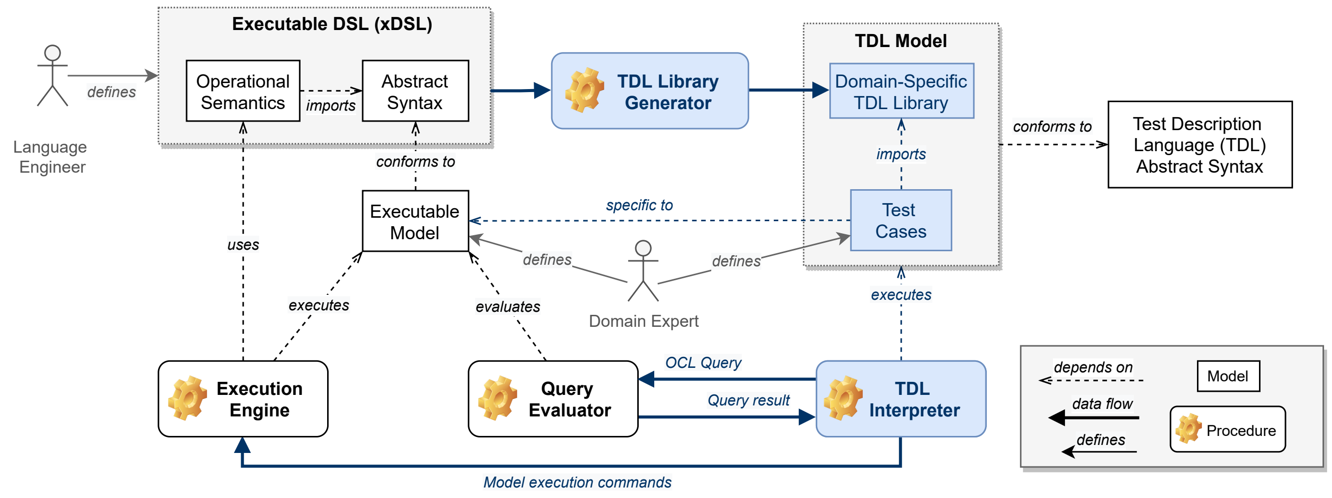


**Figure 4** Overview of the Proposed Approach

the Execution Engine is used by the TDL Interpreter to start the execution of a model, to get the content of the model, or to set the model in a specific runtime state. We also assume that the *Query Evaluator* can evaluate Object Constraint Language (OCL) queries. The Query Evaluator is used by the TDL Interpreter to evaluate queries written inside TDL test cases, so the query validation result can be used as part of the oracle of a TDL test case.

With everything in place, the domain expert can use the generated domain-specific TDL library to write test cases, and can then use the TDL interpreter to execute these test cases. We provide a detailed explanation of the *TDL Library Generator* and the *TDL Interpreter* in Sections 4 and 5.

## 4. TDL Library Generator

This section presents the *TDL Library Generator*, whose role is to produce a TDL library specific to a given xDSL. We first present how the component works and then explain what are the contents of the generated TDL library. Afterward, we show how such a library can be used by the tester (i. e., the domain expert) to write test cases for models conforming to the xDSL.

### 4.1. Description of the Generator Logic

The output of the generator is a TDL library providing a set of TDL elements for the tester. The first set of generated TDL elements are the required data types for the definition of test data. These data types are provided through a model transformation which uses the Ecore metamodel of the xDSL as input and produces a set of xDSL-specific TDL data types as output. The second set of generated TDL elements are necessary to enable the TDL test cases to request the execution of the model under test, to write OCL queries in the TDL test cases, and to configure the TDL test cases for their execution. These elements are provided by a code generator which only requires the name of the xDSL as input.

### 4.2. Description of the Generated TDL Library

Given an xDSL, three TDL packages are produced by the generator: the *xDSL-Specific Types* package, the *common* package, and the *Test Configuration* package.

**xDSL-Specific Types Package** As discussed in Section 2.2, to use TDL for a specific domain, all data types required for the specification of test data have to be provided to the tester. Yet, in the context of testing executable models, we can observe that these data types are in fact already defined as part of the definition of the considered xDSL, both in the abstract syntax and in the definition of the possible runtime states in the operational semantics. It is therefore possible to infer the required TDL data types from this information.

As explained in Section 2.1, we consider that the Ecore metamodel of the xDSL includes both the abstract syntax and the definition of the possible runtime states of conforming executed models. More precisely, runtime states are defined as additional metamodel features decorated with a "dynamic" annotation. Accordingly, we rely on a model transformation from Ecore to TDL to automatically generate all the required TDL data types for testing executable models conforming to a given xDSL.

```
rule concreteInheritedClass2structuredType {
  from class: Ecore!EClass (
  class.eAllStructuralFeatures.notEmpty()
  and class.eSuperTypes.notEmpty()
  and not class.abstract)
  to type: TDL!StructuredDataType(
  name ← class.name,
  member ← class.eStructuralFeatures→
  collect(f | if (f.isDynamicFeature)
  then thisModule.
  dynamicFeature2annotatedMember(f)
  else thisModule.staticFeature2member(f)
  endif),
  extension ← thisModule.superClass2extension(
  class))
}
```

**Listing 3** Example of an Ecore to TDL Transformation Rule

**Table 1** Outline of the Ecore to TDL Transformation Rules

| **Source Ecore element** | **Target TDL element** |
|---|---|
| EClass with no EStructuralFeature | SimpleDataType |
| EClass containing EStructuralFeature | StructuredDataType containing one Member per EStructuralFeature |
| Abstract EClass | an Annotation named 'abstract' is set to its corresponding DataType |
| Inherited EClass | an Extension of the DataType generated for its last super class, assigns to its related DataType |
| EStructuralFeature (EAttribute and EReference) | Member contained in the StructuredDataType that is generated for its container EClass. Its type is the DataType corresponding to the feature eType |
| Dynamic EStructuralFeature[3] | an Annotation named 'dynamic' is set to its corresponding Member |
| EDataType | SimpleDataType |
| EEnum | SimpleDataType |
| EEnumLiteral | SimpleDataInstance that its type is the SimpleDataType of the related EEnum |
| EPackage | Package containing all the generated elements as packagedElement |

A summary of the transformation rules is shown in Table 1. Each rule takes one element from the left column and transforms it into an element of the same row of the right column. In a nutshell, the objective of this transformation is to transform each Ecore class into a TDL data type, either simple or structured. An Ecore class with structural features (i. e., attributes and references) is transformed to a structured data type containing members, each of which corresponds to one feature of the class. To distinguish abstract classes from concrete ones and dynamic features from static ones, annotations are generated and assigned to the corresponding element. An inheritance relationship between two classes is transformed into an extension relationship in TDL. The transformation generates simple TDL data types for Ecore primitive data types and enums, and enum literals are transformed to the instances of the TDL data type corresponding to their related enum. Finally, an Ecore package is transformed to a TDL package that is the root container of all the generated TDL elements.

We used the ATL language (Jouault et al. 2006) to define the transformation rules. An excerpt of this transformation is shown in Listing 3, with the ATL code for transforming a concrete Ecore class containing features and superclasses, to a TDL structured data type containing members and an extension to the last super class. This rule has several calls to other transformation rules that are not shown here (the source code of the complete ATL transformation is available here).

Listing 4 presents the TDL DataTypes generated from the Ecore metamodel of the xFSM DSL (part **a** of Figure 1). The abstract Annotation element (line 2) is for specifying

```
1  Package fsmSpecificTypes {
2    Annotation abstract;
3    Annotation dynamic;
4    Type NamedElement (
5      _name of type EString
6    ) with {abstract;};
7    Type StateMachine extends NamedElement (
8      ownedStates of type State,
9      initialState of type State,
10     ownedTransitions of type Transition,
11     currentState of type State with {dynamic;},
12     unprocessedString of type EString
13       with {dynamic;},
14     consumedString of type EString
15       with {dynamic;},
16     producedString of type EString
17       with {dynamic;}
18   );
19   Type State extends NamedElement (
20     owningFSM of type StateMachine,
21     outgoingTransitions of type Transition,
22     incomingTransitions of type Transition
23   );
24   Type Transition extends NamedElement (
25     source of type State,
26     target of type State,
27     input of type EString,
28     output of type EString
29   );
30 }
```

**Listing 4** TDL Data Types generated for the xFSM DSL

the TDL DataTypes generated for an abstract class, such as NamedElement (lines 4-6). Moreover, to distinguish the TDL DataTypes generated for the dynamic elements, an Annotation named dynamic (line 3) is defined and assigned to them, such as the currentState member of the StateMachine (line 11).

**Common Package** This package contains common elements that are not specific to the given xDSL, but provide common testing facilities for any xDSL. As shown in Listing 5, the Verdict Type (line 2) along with several instantiations of it (lines 3-5) are defined to be used for test verdict assignment. This Package also provides elements for performing several operations on the model under test (later referred to as *model execution commands*), including runModel for executing the

```
1  Package common {
2    Type Verdict ;
3    Verdict PASS ;
4    Verdict FAIL ;
5    Verdict INCONCLUSIVE ;
6
7    Type modelExecutionCommand ;
8    modelExecutionCommand runModel ;
9    modelExecutionCommand resetModel ;
10   modelExecutionCommand getModelState ;
11
12   Type OCL ( query of type EString ) ;
13   OCL oclQuery ( query = ? );
14 }
```

**Listing 5** TDL Common Package for all xDSLs

[3] it has an EAnnotation element named as 'dynamic'

```
1   Package testConfiguration {
2     Import all from common;
3
4     Gate Type genericGateType accepts
      modelExecutionCommand;
5     Gate Type oclGateType accepts OCL;
6     Component Type TestSystem having {
7       gate genericGate of type genericGateType;
8       gate oclGate of type oclGateType;
9     }
10    Component Type MUT having {
11      gate genericGate of type genericGateType;
12      gate oclGate of type oclGateType;
13    }
14
15    Annotation MUTPath;
16    Annotation DSLName;
17
18    Test Configuration fsmConfiguration {
19      create Tester tester of type TestSystem;
20      create SUT fsm of type MUT with {
21        MUTPath:'TODO: Put the path to the MUT';
22        DSLName:'TODO: Put the name of the DSL';
23      };
24      connect tester.genericGate to fsm.
      genericGate;
25      connect tester.oclGate to fsm.oclGate;
26    }
27  }
```

**Listing 6** TDL Test Configuration Package generated for the xFSM DSL

model (line 8), `resetModel` for resetting its state to the default (line 9), and `getModelState` for getting its current state, i. e., the content of its dynamic features (line 10). To enable the tester to use OCL queries in the test cases, this package provides a data type named `OCL` (line 12) and an instantiation of it (line 13). This instantiation uses the question mark TDL symbol (`?`) for the `query` attribute, which means that a value must be given to this attribute when the `oclQuery` instance is used.

**Test Configuration Package** In TDL, a test case must refer to a *test configuration* defining what is the system under test, and how to communicate with it. In particular, a test configuration can define what are the available communication *gates*, each gate allowing specific types of messages. In the present approach, we consider that two kinds of messages can be exchanged with the model under test: *model execution commands* related to the execution of the model, and *OCL commands* related to the execution of the OCL queries. Accordingly, given an xDSL, our generator will generate a TDL Test Configuration Package introducing these gates and components for the xDSL, along with a test configuration that makes use of them.

Listing 6, shows the `testConfiguration` package generated for the xFSM DSL. It defines two types of gates: `genericGateType` for model execution commands (line 4), `oclGateType` for OCL commands (line 5). These gates are used by two types of components: `TestSystem` to represent the test system i. e. the test suite itself (lines 6-9) and `MUT` to represent the model under test (lines 10-13). Lastly, the test configuration `fsmConfiguration` is defined, which creates both

```
1   Package bitShiftingFSM_TestSuite {
2     Import all from common;
3     Import all from fsmSpecificTypes;
4     Import all from testConfiguration;
5
6     StateMachine stateMachineNewState(
7       _name = "BitShifting");
8     State S2 (_name = "S2");
9
10    Test Description bitShiftingGenericTest uses
      configuration fsmConfiguration{
11      tester.genericGate sends
      stateMachineNewState
12        (unprocessedString = "10010110")
13        to fsm.genericGate;
14      tester.genericGate sends runModel
15        to fsm.genericGate;
16      tester.genericGate sends getModelState
17        to fsm.genericGate;
18      fsm.genericGate sends stateMachineNewState
19        (producedString = "01001011")
20        to tester.genericGate;
21    }
22    Test Description bitShiftingOclTest uses
      configuration fsmConfiguration{
23      tester.genericGate sends
      stateMachineNewState
24        (unprocessedString = "000101010")
25        to fsm.genericGate;
26      tester.genericGate sends runModel
27        to fsm.genericGate;
28      tester.oclGate sends oclQuery
29        (query = "self.currentState")
30        to fsm.oclGate;
31      fsm.oclGate sends S2 to tester.oclGate;
32    }
33  }
```

**Listing 7** Two sample TDL test cases for the bit shifting xFSM: (1) an executable version of the test case of Listing 2 (2) using an OCL query in the test case

components and connects their gates (lines 18-26). In addition, we rely on TDL annotations to define additional information inside the MUT component: `MUTPath` containing the path to the file containing the model to test (line 21), and `DSLName` containing the unique identifier of the DSL (line 22).

## 4.3. Using the TDL Library to write Test Cases

Listing 7 presents two test cases for the bit shifting FSM model (previously shown in Figure 3). The first test case (lines 10-21) is a more complete and executable version of the example test case previously introduced in Listing 2. Contrary to Listing 2, all the required data types and test configurations for writing test cases are provided by the generated TDL library and are obtained by import declarations. Using the data types provided by the `fsmSpecificTypes` package (line 3), the domain expert can define runtime states for the bit shifting FSM by instantiating the `StateMachine` type (`stateMachineNewState` in lines 6-7) and setting its dynamic features in the test case description (line 12 as test input data and line 19 as expected output). In the first test case, the tester component first sends the test input data—that is the bit shifting FSM in a runtime state where its unprocessedString is set as "10010110" (line 12)—to the tested model, i. e., the bit shifting FSM. By using the model execution commands provided by the commonPackage (line 2), the tester component then requests a run of the model (lines 14-15), and also gets the current runtime state of the model (lines 16-17) to be asserted against the expected output data—the bit shifting FSM in a new runtime state where its producedString is "01001011" (line 19).

In the second test case, we use OCL queries to check the value of specific dynamic features using the elements provided by the commonPackage (line 2). After sending the test input data (lines 23-25), and requesting for running the bit shifting FSM in the specified runtime state (lines 26-27), the tester component sends an OCL query (line 29) through the OCL gates to get the value of the currentState of the bit shifting FSM after its execution (lines 28-30). Finally, the expected result, i. e., S2 (line 31) will be checked against the query evaluation result to set the test case verdict as pass or fail. As is shown, by using the domain-specific data types provided by the fsmSpecificTypes package, the domain expert can define the model elements to use them as test data, such as the S2 of type State (line 8) that is used when defining the oracle (line 31).

## 5. TDL Operational Semantics for xDSLs

In this section, we present an operational semantics for TDL tailored for the testing of executable models. We initially present how we refined the execution semantics provided in the TDL standard for the testing of executable models, We then define the operational semantics itself, and explain how it is decoupled from both the xDSLs and the metaprogramming approaches used for their implementation.

### 5.1. Adapting the TDL semantics to model execution

The TDL Standard[4] consists of several documents: the TDL metamodel, the TDL graphical syntax, the TDL exchange format, the UML profile for TDL, the mapping from TDL to TTCN-3, and two extensions that we will not discuss here.

Two parts of the standard are related to the semantics of TDL. First, the *metamodel* document (ETSI ES 203 119-1 2020) specifies the abstract syntax as a metamodel, and its associated semantics using natural language. It introduces the basic principles of TDL and describes all the test-specific concepts included in the TDL abstract syntax, categorized as Foundation, Data, Time, Test Configuration, and Test Behavior. For each concept, the semantics, the relationship with other concepts, the properties, and the static constraints are also provided. Second, the *mapping to TTCN-3* document is essentially a translational semantics for TDL using TTCN-3 as a target language. While this semantics does allow the execution of TDL test cases, it is only partial and mainly aims to manage test cases for software systems communicating through common protocols (TCP, UDP, SQL, HTTP, etc.). Therefore, we deemed this translational semantics too distant from the aim of this work—i. e. the testing of executable models. Instead, we solely used the execution semantics described in the *metamodel* document as the reference specification for the operational semantics we propose.

[4] https://tdl.etsi.org/index.php/downloads

However, as previously explained, our approach aims to cover two additional concerns: managing the execution of models, and managing OCL queries. As these concerns are not covered by the standardized TDL semantics, we have to make adaptations for the operational semantics we propose. In Section 4, the *TDL Library Generator*, takes these concerns into account by generating specific *data types* (ModelExecutionCommand and OCL in the common package) along with *gate types* that accepting data conforming to them (genericGate and oclGate in the testConfiguration package). Accordingly, our operational semantics must be able to interpret such generated elements. This will be presented in the next subsection.

In addition, to specify the result of a test case execution, we support the verdicts provided by the TDL *metamodel* document, including PASS, FAIL, and INCONCLUSIVE (ETSI ES 203 119-1 2020). PASS and FAIL correspond to observing valid and invalid behaviors of the SUT, respectively, while INCONCLUSIVE is used when neither pass nor fail can be assigned. For instance, if the *tester* sends a syntactically wrong OCL query to the *SUT*, the TDL Interpreter will interrupt the test case execution and the verdict will be assigned as INCONCLUSIVE.

It should be noted that at the moment, we do not support all the TDL elements, such as Time and complex Behavior concepts (e. g., Parallel, Exceptional, Periodic). These concepts enable the tester to define different types of tests such as load tests and distributed tests, so we consider them as future work.

### 5.2. Definition of the TDL Interpreter

We call *TDL Interpreter* the component that embodies the TDL operational semantics we propose. Algorithm 1 shows the main loop of the interpreter. Its input is a TDL package that contains TDL test cases. Each test case uses a specific configuration that must be activated first to enable the test case execution (line 3). It also contains a set of behaviors of different types, each of which has a different operational semantics. For instance, to execute a behavior of type Message (line 5), according to its source gate, indeed the role of its container component, the argument is treated differently. When it has the SUT role, the argument is the expected result that has to be asserted (line 10) and when the role is the Tester, the argument is the test input data that has to be sent to the model under test (line 13).

### 5.3. Connection to the External Components

We assume that the *TDL Interpreter* is connected to two external components: the *Execution Engine* and the *OCL Query Evaluator*. First, the Execution Engine must provide services to manage the execution of the model under test (e. g., running the model, resetting its state to default, getting its current state), and is used by the TDL Interpreter to interpret all the model execution commands used in a test case. Second, the OCL Query Evaluator must provide services to trigger the execution of an OCL query on the model and to retrieve the result.

As can be seen in Figure 4, the execution engine uses the operational semantics of the xDSL to execute its conforming models. Since the operational semantics of xDSLs can be de-

**Algorithm 1:** The algorithm of the TDL Interpreter main loop

```
Input: package: the TDL package containing the TDL test cases to be
       executed
begin
    foreach $testcase \in package.testCases$ do
        $testcase.configuration$.activate()
        foreach $behavior \in testcase.behaviors$ do
            if behavior is Message then
                $sourceGate \leftarrow behavior.source$
                $targetGate \leftarrow behavior.target$
                if sourceGate.component.role is SUT then
                    $testOracle \leftarrow behavior.argument$
                    $targetGate$.assert($testOracle$)
                else if sourceGate.component.role is Tester then
                    $testInputData \leftarrow behavior.argument$
                    $targetGate$.sendDataToSUT($testInputData$)
            else if behavior is <other behavior types> then
                ...
```

fined using different *metaprogramming approaches*—i. e. one or several metalanguages used in a particular fashion— various execution engines are required. This in turn reveals one source of heterogeneity of xDSLs that is supported by our *generic* approach as follows. For running tests on models conforming to a given xDSL, the TDL Interpreter automatically selects the appropriate execution engine (i. e., the engine dedicated to the metaprogramming approach used for implementing the operational semantics of the given xDSL). This is done when activating the test configuration of a running test case (line 3 of Algorithm 1). Indeed, according to the `DSLName` specified in the `configuration` of the `testcase`, the TDL Interpreter looks for an appropriate engine from the existing ones. It can then start the engine and make a request for running the model in different states specified in the test cases (i. e., model execution commands). This communication takes place in line 13 of Algorithm 1. If the `argument` of the `behavior` is a model execution command, it will be sent to the model through the execution engine that was previously started in line 3. Therefore, as long as there is an execution engine dedicated to a specific metaprogramming approach, our proposed approach can use it for running tests on models conforming to any xDSL whose operational semantics is defined using that specific metaprogramming approach.

Besides, to evaluate the OCL queries specified in the TDL test cases on the model, the interpreter has to communicate with an OCL Query Evaluator. To this end, if the `configuration` of the `testcase` includes an `oclGate`, the OCL Query Evaluator will be initialized when activating the test configuration (line 3 of Algorithm 1). Consequently, whenever the `argument` of the `behavior` is an OCL query, it will be sent to the model through the query evaluator that was previously initialized in line 3.

To provide the mentioned communications while decoupling the TDL Interpreter from various execution engines, we rely on an specific interface for execution engines. Figure 5 shows the ExecutionEngine interface and the dependencies of the TDL

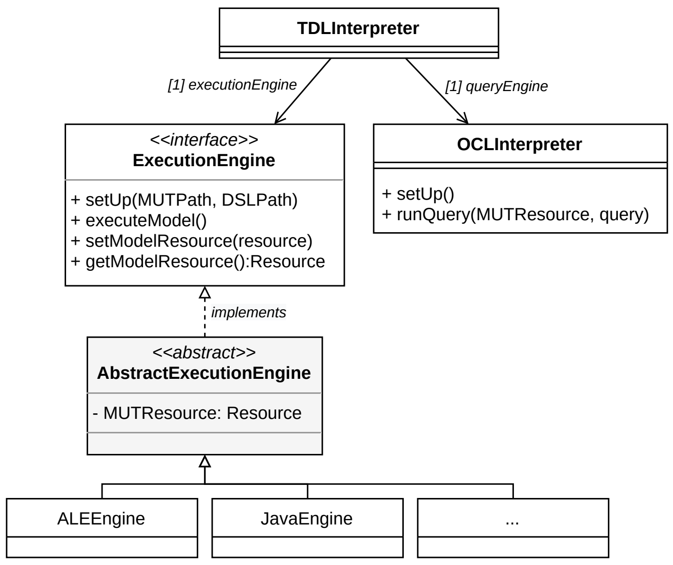


**Figure 5** Class diagram showing the main dependencies of the TDL interpreter

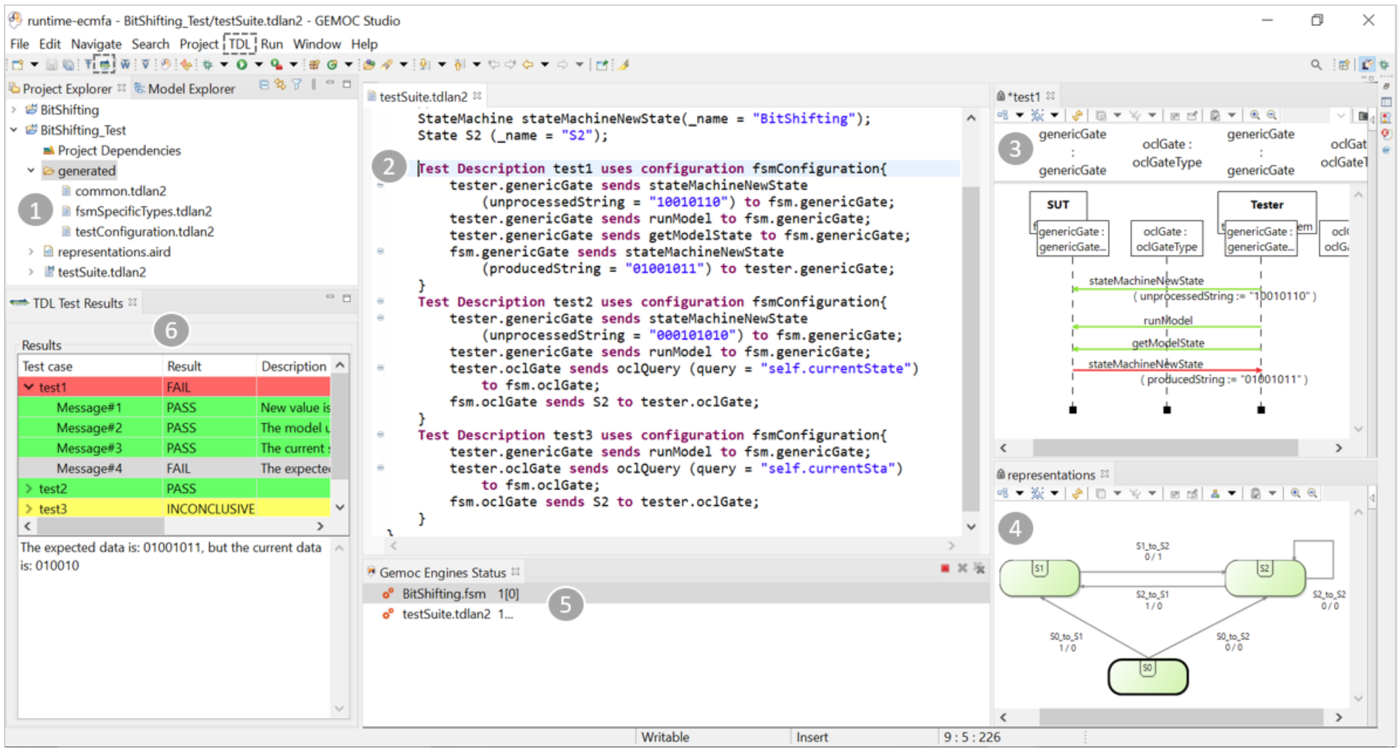
Interpreter using a class diagram. Following the terminology of popular frameworks such as EMF, we call *resource* the artifact that contains the model to load and execute. A resource may be a file, or a URL to access the model remotely, or a database connection. The ExecutionEngine interface contains methods for setting up the execution engine based on the model under test (i. e., the model to be executed) and its conforming xDSL, executing the model, setting the model in a specific state, and getting its current state. The ExecutionEngine interface is partially implemented by the AbstractExecutionEngine class, and then further specialized for each metaprogramming approach (see Section 6 for examples). Therefore, our proposed approach is not restricted to a specific execution engine, but can support all the existing ones. Finally, we also rely on a specific interface for the OCL Query Evaluator, here with the class named OCLInterpreter.

## 6. Tool Support

We implemented our approach as part of the GEMOC Studio (Bousse et al. 2016), a language and modeling workbench defined on top of the Eclipse Modeling Framework (EMF). TDL is also implemented using EMF technologies, making it easier to make both work together.

We used the ATL transformation language (Jouault et al. 2006) for implementing the Ecore to TDL transformation. For the TDL interpreter, we used Xtend (Efftinge et al. 2012) to implement the execution rules of the TDL operational semantics. To evaluate the OCL queries, we used the Eclipse OCL API (Damus et al. 2020).

The GEMOC Studio supports several metaprogramming approaches. This includes Java-based languages (Kermeta (Jézéquel et al. 2015), Xtend (Efftinge et al. 2012), and pure Java), the Action Language for EMF (ALE) (Leduc et al. 2017), xMOF (Mayerhofer et al. 2013), and a combination of a Java-based language with the MoCCML language (Deantoni et



```
    StateMachine stateMachineNewState(_name = "BitShifting");
    State S2 (_name = "S2");

    Test Description test1 uses configuration fsmConfiguration{
        tester.genericGate sends stateMachineNewState
            (unprocessedString = "10010110") to fsm.genericGate;
        tester.genericGate sends runModel to fsm.genericGate;
        tester.genericGate sends getModelState to fsm.genericGate;
        fsm.genericGate sends stateMachineNewState
            (producedString = "01001011") to tester.genericGate;
    }
    Test Description test2 uses configuration fsmConfiguration{
        tester.genericGate sends stateMachineNewState
            (unprocessedString = "000101010") to fsm.genericGate;
        tester.genericGate sends runModel to fsm.genericGate;
        tester.oclGate sends oclQuery (query = "self.currentState")
            to fsm.oclGate;
        fsm.oclGate sends S2 to tester.oclGate;
    }
    Test Description test3 uses configuration fsmConfiguration{
        tester.genericGate sends runModel to fsm.genericGate;
        tester.oclGate sends oclQuery (query = "self.currentSta")
            to fsm.oclGate;
        fsm.oclGate sends S2 to tester.oclGate;
    }
}
```

Results

| Test case | Result | Description |
|---|---|---|
| test1 | FAIL | |
| Message#1 | PASS | New value is |
| Message#2 | PASS | The model u |
| Message#3 | PASS | The current : |
| Message#4 | FAIL | The expecte |
| test2 | PASS | |
| test3 | INCONCLUSIVE | |

The expected data is: 01001011, but the current data is: 010010

**Figure 6** A screenshot of the TDL testing facilities running in the GEMOC Studio modeling workbench for the bitShifting xFSM

al. 2015). Each of these approaches is supported by a dedicated execution engine. Hence, a significant part of the implementation of the TDL Interpreter is dedicated to managing all these execution engines properly. For instance, the TDL Interpreter must read the xDSL definition in a GEMOC-specific `.dsl` file to discover the used metaprogramming approach, in order to start the correct execution engine accordingly.

Figure 6 shows a screenshot of the resulting tool. The source code is available on a public GitLab instance[5]. There are GUI icons in the toolbar and the menu bar allowing the user to choose an xDSL and run the generator for it. To use the tool for testing the bit shifting FSM model (label 4), we initially run the generator for the xFSM DSL and the generated TDL packages can be seen in the Project Explorer (label 1). Here we show the execution of a test suite containing three sample TDL test cases (label 2), two of which already described in Listing 7. The graphical representation of the first test case is shown on the top right side (label 3). We extended the TDL graphical syntax to color the arrow corresponding to each message based on the result of its execution when executing the test case. Accordingly, if the message is successfully executed, the arrow will be highlighted to green and otherwise to red. The status of the GEMOC engines (label 5) demonstrates that the model and its test suite are running. We provide a view to report the test execution result along with some useful information for the user (label 6). As can be seen, the first test case has failed, the second has passed, and the result for the third one is inconclusive since the OCL query is syntactically wrong and cannot be created.

[5] https://gitlab.univ-nantes.fr/naomod/faezeh-public/xtdl

## 7. Evaluation

We designed and performed an empirical evaluation of our approach to consider its *genericity* by answering the following research questions:

**RQ#1** Does the approach provide testing facilities for xDSLs in which their abstract syntax is designed for *different domains*?

**RQ#2** Does the approach provide testing facilities for xDSLs in which their operational semantics is implemented using *different metaprogramming approaches*?

**Considered xDSLs:** We tried our testing approach for five xDSLs, covering different domains including FSM, Arduino, and BPMN [6] to answer RQ#1. For the FSM and the Arduino languages, we used two different implementations of their semantics (i. e., implemented using Kermeta and ALE metaprogramming approaches) to answer RQ#2. We used the FSM xDSL with Kermeta semantics and with ALE semantics (i. e., already described in Section 2) that are both provided by the GEMOC language workbench. We also used the open source project of the Arduino xDSL with Kermeta semantics, and then we implemented its semantics using the ALE language to have two different implementations of the Arduino xDSL. We made minor modifications in these existing xDSLs to match the assumptions described in Section 2.1.

As the last xDSL used in our evaluation, we have defined BPMN xDSL that is a representative of the *Microflow DSL*[7], a real-world interpreted language introduced by the Mendix development platform. Microflow DSL enables the non-technical domain experts to define the application logic throug h data-flow

[6] Business Process Model and Notation
[7] https://docs.mendix.com/refguide/microflows

**Table 2** The Evaluation Data

| xDSL | DSL size | | Model | Model size (n. of EObjects) | TDL Library Size (LoC generated) | Test suite size | |
|---|---|---|---|---|---|---|---|
| | Abstract syntax size (n. of EClasses) | Semantics size (LoC) | | | | n. of Test cases | LoC |
| xFSM | 3 | K3: 110 ALE: 90 | TrafficLight | 7 | 111 | 4 | 31 |
| | | | BitShifting | 9 | | 4 | 33 |
| | | | EdgeDetector | 9 | | 8 | 58 |
| | | | ToLowerCase | 133 | | 5 | 39 |
| | | | ToUpperCase | 133 | | 5 | 39 |
| xArduino | 59 | K3: 667 ALE:421 | Servo9g | 15 | 259 | 4 | 35 |
| | | | ActiveWaitIR | 18 | | 4 | 38 |
| | | | TurnOnLED | 18 | | 2 | 26 |
| | | | ServoIrButton | 59 | | 4 | 65 |
| xBPMN | 39 | ALE: 318 | VerifyUserAccess | 26 | 202 | 2 | 44 |
| | | | PromoteEmployee | 46 | | 4 | 60 |

modeling. It contains elements to perform CRUD operations on data objects, to show UI pages, to make choices, etc, and its graphical syntax is based on the BPMN standard. Since its source code is not open to access, we re-implemented it from scratch. The design of our BPMN xDSL conforms to the definitions presented in Section 2.1 and its semantics is implemented using the ALE language. The definition of all five xDSLs are available in a public git repository.

**Workflow:** We applied our testing approach on each considered xDSL, following the same workflow. In reference to the approach overview depicted in Figure 4, we initially executed the *TDL Library Generator* component. The component generated a TDL library specific to each given xDSL, successfully. Since our approach provides testing facilities for models conforming to a given xDSL, we prepared a set of models under test (MUTs) conforming to each considered xDSL. Afterward, we defined test cases for each MUT using the *model execution commands* and the *OCL queries* provided by the generated TDL Library. Finally, all the test cases were executed on their related MUT, the test verdicts were set, and the test results were reported through the graphical view provided by our tool. All the tested models and their test cases are publicly accessible on a public GitLab instance.

**Evaluation result:** Table 2 presents all the information related to our evaluation including 1) the size of each xDSL as the number of the EClasses of its abstract syntax and the number of lines of code (LoC) of its semantics; 2) the tested models with their size in terms of the number of their EObjects; 3) the number of LoC of the TDL library automatically generated for each considered xDSL; and 4) the test suite size as the number of the test cases per tested model along with the number of LoC manually written by the tester. Most noticeably of all, it can be seen that, unsurprisingly, the size of the generated TDL Library increases with the size of the xDSL. This highlights one benefit of using our approach since the generated library provides all the TDL boilerplate code that the domain expert would otherwise write by hand. Therefore, the proposed approach reduces the cost of providing testing support for a given xDSL.

Successful use of the approach for testing different executable models conforming to various xDSLs, demonstrates that it satisfies the *genericity* aspect.

**Discussion:** Our experiment on the Arduino xDSL reveals some limitations of the proposed testing approach. Using this xDSL, one can define time-dependent behaviors in the Arduino models, such as turning on an LED after 200 milliseconds. It is also possible to define Arduino models that continuously collect data from the outside of the model (e. g., the ambient temperature) and react to specific event occurrences (e. g., high rise and fall of the ambient temperature). To test the behavior of such models, the test cases should be able to communicate with the model under test during its execution. For example, sending test data to the running model at a certain time and asserting if the model reacts as expected.

More precisely, to provide testing support for the xDSLs whose contain time-related concepts and/or have an event-driven behavior—called reactive DSLs—, specific requirements must be considered. In models conforming to reactive xDSLs, some behaviors can happen at a specific time or under reception of specific event occurrences from the outside of the model *during* its execution. For testing these kinds of behaviors, one possibility could be to support at least two new commands in TDL test cases: (1) controlling the execution of the tested model by requesting for the *asynchronous* execution of the model, waiting for a precise time duration, and requesting to stop the model execution; and (2) interacting with the tested model during its execution through the exchange of the event occurrences. There are some existing work trying to define the characteristics of the reactive DSLs and their specific testing requirements (Meyers et al. 2016; Leroy et al. 2020) which would be considered in our future work.

## 8. Related Work

Testing support for DSLs is already investigated but in different paths. While some efforts are interested in defining support for a specific DSL, others propose generic solutions for different DSLs. In this section, we provide an overview of the related work in both paths.

### 8.1. Ad-hoc Testing Facilities for DSLs

Sequencer Testing Tool (SeTT) is a testing framework proposed particularly for the Sequencer DSL (Kos et al. 2016), a DSL for data acquisition included in the NASA awarded measurement system DEWESoft for adjusting measurements and creating measurement procedures. SeTT is the result of extending Sequencer DSL with basic testing concepts such as assertions. Using SeTT, the domain expert defines test cases for each part of the measurement system by augmenting the model under test with test elements.

A BPMN-based testing approach is proposed in (Lübke & van Lessen 2017) to facilitate the definition of test cases for executable business processes (i. e., described in WS-BPEL or BPMN2). They define a test profile for BPMN by adding testing concepts such as *Assertions* to the BPMN metamodel. Some control-flow restrictions are also enforced to ensure determinism of test models. A BPMN test model contains several process *Pools*, one for the process under test and the others for the tests, which communicate through message exchange. The mapping between the test case model and the physical operations (i. e., in the process under test), and the technical information for their execution must be previously defined.

Mijatov et al. propose a functional testing framework for a subset of fUML[8] including class and activity diagrams (Mijatov et al. 2015). The framework introduces an executable test specification language to be used for describing and running test cases that control the behavior of fUML models. Test cases use temporal expressions for the precise selection of the runtime states that need to be asserted. Since fUML models can describe a system with concurrent behavior, OCL queries are also supported for specifying complex assertions on the runtime states of a system that behaves concurrently. In addition, they propose a new algorithm for the verification of execution order of activity nodes of concurrent systems.

A simulation and test generation approach for fUML activity diagrams is proposed in (Iqbal et al. 2019a). The approach is mainly based on a translation from fUML Activity diagram containing Alf[9] code to Java. The generated Java code is used for the automatic generation of test input data required for an exhaustive simulation of the fUML model along with its associated Alf code, as well as automatic generation of test cases with oracle based on the provided simulation. The generated test cases satisfy 100 % coverage of the Java code.

Juhnke and Tichy define a domain analysis method for automotive systems test specification (Juhnke & Tichy 2019). The method applies to the analysis phase of the DSL development, to extract the domain-specific concepts from the existing textual test specifications of a specific automotive system. The concepts are then used when generating a specific test DSL for describing logical test cases (acceptance and customer experience test cases), which will be implemented or executed manually on the hardware system or the prototype vehicle.

To sum up, ad-hoc testing solutions promote usability, as they enable the domain experts to describe test cases using the system description language that is familiar to them. Nevertheless, they are mainly defined by adding basic testing elements such as *Assertion* into the main DSL to support testing at the unit level, hence they are not adapted for different levels of testing, such as integration. Moreover, they lack reusability, since a new test language must be engineered for each new DSL.

### 8.2. Generic Testing Facilities for DSLs

Wu et al. propose a generic unit testing framework for grammar-based xDSLs, whose semantics is translated to a GPL that offers a unit testing framework (e. g., JUnit for Java) (Wu et al. 2009). The translator of the framework keeps the traceability links between the DSL code and its corresponding generated GPL code, and the language engineer must define the mapping algorithms between the testing actions of DSL and those of the GPL. At first, the framework translates DSL test cases into the GPL, allowing the GPL testing tool to execute them on the compiled code of the model under test (i. e., the GPL code). Afterward, the framework translates the test results in the GPL level back to the DSL level and presents them to the domain expert. Therefore, these two sets of mappings enable using of GPL testing tools for testing executable models conforming to the compiled DSLs.

Meyers et al. propose a generic testing approach for xDSLs with operational semantics (Meyers et al. 2016). They use a DSL extension mechanism that augments a given xDSL with a limited set of testing features to automatically generate a specific test language for the given xDSL. To execute test cases, the operational semantics of the input xDSL has to be instrumented for each single test case, separately. Therefore, a variant of the operational semantics must be generated for individual test cases. In the instrumentation, a call to a new execution rule is added between each execution rule of the xDSL's semantics. Consequently, the new rules have to be implemented in the same metaprogramming approach as the execution rules of the semantics of the xDSL.

Contrary to them, our approach is independent of the metaprogramming approach used for defining the operational semantics of the xDSL. Additionally, we use a standard testing language, TDL, covering more than 100 testing concepts in its abstract syntax (such as 20 different types of test behaviors), while they defined a custom test metamodel with 20 concepts.

## 9. Conclusion and Future Work

Providing testing facilities for any given xDSL is a challenging task concerning the diversity of xDSLs. This diversity originates from both, the domain described by the xDSL abstract syntax and the approach used for the implementation of its semantics. In this paper, we proposed a generic testing approach

[8] Foundational Subset for Executable UML Models
[9] Action language for fUML

for metamodel-based xDSLs. It uses the TDL standard testing language for describing test cases and provides solutions to specialize TDL for testing executable models conforming to xDSLs. Indeed, we proposed a TDL library generator that generates a domain-specific TDL library for a given xDSL, allowing the domain expert to write test cases for testing the executable models conforming to it. We also provided an interpreter for TDL to execute TDL test cases on executable models. Our evaluation on 5 xDSLs demonstrates that the approach realizes the genericity aspect, and thus is reusable by various xDSLs.

Perspectives for future work include both, applying the approach on more xDSLs and extending it with more features. We plan to explore the approach on more complex xDSLs (having many dynamic features) and consequently, on models with more complex runtime states. For extending the proposed approach in the future, there are several interesting lines of research. At the moment, we only support testing of executable models conforming to one xDSL, so one possible extension is supporting models conforming to several interconnected xDSLs. In Section 5.1, we noted that the TDL interpreter does not support all the elements of TDL standard abstract syntax currently. By completing the definition of TDL Interpreter, we can write and execute more complex TDL test cases to control different features of the model under test (not just its functionality).

**Acknowledgments**

This project has received funding from the European Union's Horizon 2020 research and innovation program under the Marie Skłodowska Curie grant agreement No 813884.

## References

Bendraou, R., Combemale, B., Crégut, X., & Gervais, M.-P. (2007, December). Definition of an eXecutable SPEM 2.0. In *14th Asia-Pacific Software Engineering Conference (APSEC)* (p. 390-397). Nagoya, Japan: IEEE Computer Society.

Bousse, E., Degueule, T., Vojtisek, D., Mayerhofer, T., Deantoni, J., & Combemale, B. (2016). Execution framework of the gemoc studio (tool demo). In (p. 84–89). New York, NY, USA: Association for Computing Machinery.

Damus, C., Sánchez-Barbudo Herrera, A., Uhl, A., & Willink, E. (2020). *Ocl documentation* (Tech. Rep.). Retrieved from https://download.eclipse.org/ocl/doc/6.12.0/ocl.pdf

Deantoni, J., Diallo, I. P., Teodorov, C., Champeau, J., & Combemale, B. (2015). Towards a meta-language for the concurrency concern in dsls. In *2015 design, automation & test in europe conference & exhibition (date)* (pp. 313–316).

Efftinge, S., Eysholdt, M., Köhnlein, J., Zarnekow, S., von Massow, R., Hasselbring, W., & Hanus, M. (2012, September). Xbase: Implementing domain-specific languages for java. *SIGPLAN Not.*, *48*(3), 112–121.

ETSI ES 203 119-1. (2020). *Methods for testing and specification (mts); the test description language (tdl); part 1: abstract syntax and associated semantics.* Retrieved from https://tdl.etsi.org/index.php/downloads

Fischer, T., Niere, J., Torunski, L., & Zündorf, A. (2000). Story diagrams: A new graph rewrite language based on the unified modeling language and java. In H. Ehrig, G. Engels, H.-J. Kreowski, & G. Rozenberg (Eds.), *Theory and application of graph transformations* (pp. 296–309). Berlin, Heidelberg: Springer Berlin Heidelberg.

Iqbal, J., Ashraf, A., Truscan, D., & Porres, I. (2019a). Exhaustive simulation and test generation using fuml activity diagrams. In P. Giorgini & B. Weber (Eds.), *Advanced information systems engineering* (pp. 96–110). Cham: Springer International Publishing.

Jézéquel, J.-M., Combemale, B., Barais, O., Monperrus, M., & Fouquet, F. (2015). Mashup of metalanguages and its implementation in the kermeta language workbench. *Software & Systems Modeling*, *14*(2), 905–920.

Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I., & Valduriez, P. (2006). Atl: A qvt-like transformation language. In *Companion to the 21st acm sigplan symposium on object-oriented programming systems, languages, and applications* (p. 719–720).

Juhnke, K., & Tichy, M. (2019). A tailored domain analysis method for the development of system-specific testing dsls enabling their smooth introduction in automotive practice. In *2019 45th euromicro conference on software engineering and advanced applications (seaa)* (p. 10-18).

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

Leduc, M., Degueule, T., Combemale, B., van der Storm, T., & Barais, O. (2017). Revisiting visitors for modular extension of executable dsmls. In *2017 acm/ieee 20th international conference on model driven engineering languages and systems (models)* (p. 112-122).

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

Lübke, D., & van Lessen, T. (2017). Bpmn-based model-driven testing of service-based processes. In *Enterprise, business-process and information systems modeling* (pp. 119–133). Springer.

Makedonski, P., Adamis, G., Käärik, M., Kristoffersen, F., Carignani, M., Ulrich, A., & Grabowski, J. (2019). Test descriptions with etsi tdl. *Software Quality Journal*, *27*(2), 885–917.

Mayerhofer, T., Langer, P., Wimmer, M., & Kappel, G. (2013). xmof: Executable dsmls based on fuml. In *International conference on software language engineering* (pp. 56–75).

Meyers, B., Denil, J., Dávid, I., & Vangheluwe, H. (2016). Automated testing support for reactive domain-specific modelling languages. In *Proceedings of the 2016 acm sigplan international conference on software language engineering* (pp. 181–194). Association for Computing Machinery.

Mijatov, S., Mayerhofer, T., Langer, P., & Kappel, G. (2015). Testing functional requirements in uml activity diagrams. In J. C. Blanchette & N. Kosmatov (Eds.), *Tests and proofs* (pp. 173–190). Cham: Springer International Publishing.

OASIS. (2007). *Web services business process execution language version 2.0.*

Object Management Group. (2013b). *Semantics of a foundational subset for executable uml models.*

Steinberg, D., Budinsky, F., Merks, E., & Paternostro, M. (2008). *Emf: eclipse modeling framework.* Pearson Education.

Wu, H., Gray, J., & Mernik, M. (2009). Unit testing for domain-specific languages. In W. M. Taha (Ed.), *Domain-specific languages* (pp. 125–147). Springer Berlin Heidelberg.

## About the authors

**Faezeh Khorram** is a PhD student in the NaoMod group at IMT Atlantique (France). She is also involved in the Lowcomote European project working on quality assurance in the Low-Code Development Platforms (LCDPs). Her current research interests are model testing, test-specific languages, and Domain-Specific Languages (DSLs). You can contact the author at faezeh.khorram@imt-atlantique.fr.

**Erwan Bousse** is an Associate Professor at the University of Nantes (France). He obtained his PhD in France in 2015 at the University of Rennes 1 for his work on execution traces and omniscient debugging of executable models. His current research interests include Software Language Engineering (SLE), Model Driven Engineering (MDE), Domain-Specific Languages (DSLs), model execution and simulation, and the debugging and testing of models. Contact him at erwan.bousse@ls2n.fr, or visit https://bousse-e.univ-nantes.io/

**Jean-Marie Mottu** is an Associate Professor at the University of Nantes (France). He obtained his PhD in France in 2008 at the University of Rennes 1 for his work on testing model transformations. His current research interests include Model Driven Engineering (MDE), Domain-Specific Languages (DSLs), Software Quality in particular Test Verification considering functional and non functional properties. You can contact the author at jean-marie.mottu@ls2n.fr.

**Gerson Sunyé** is an associate professor at the University of Nantes (France) in the domain of software engineering and distributed architectures and the head of the Nantes Software Modeling Group. He received the PhD degree in Computer Science from the University of Paris 6, France, in 1999. From 1999 to 2001 he was a postdoctoral researcher at the IRISA Computer Science laboratory. He has 4 years of industry experience in software development. He received his Habilitation in 2014. He is the author of several papers in international conferences and journals in software engineering. His research interests include software testing, design patterns and large-scale distributed systems. Contact him at gerson.sunye@ls2n.fr, or visit https://sunye-g.univ-nantes.io/.