Software and Systems Modeling manuscript No.
(will be inserted by the editor)

Advanced Testing and Debugging Support for Reactive Executable DSLs

Faezeh Khorram · Erwan Bousse · Jean-Marie Mottu · Gerson Sunyé

Received: date / Accepted: date

Abstract Executable Domain-Specific Languages (xDSLs) allow the definition and the execution of behavioral models. Some behavioral models are reactive, meaning that during their execution, they accept external events and react by exposing events to the external environment. Since complex interaction may occur between the reactive model and the external environment, they should be tested as early as possible to ensure the correctness of their behavior. In this paper, we propose a set of generic testing facilities for reactive xDSLs using the standardized Test Description Language (TDL). Given a reactive xDSL, we generate a TDL library enabling the domain experts to write and run event-driven TDL test cases for conforming reactive models. To further support the domain expert, the approach integrates interactive debugging to help in localizing defects, and mutation analysis to measure the quality of test cases. We evaluate the level of genericity of the approach by successfully writing, executing, and analyzing 247 event-driven TDL test cases for 70 models conforming to two different reactive xDSLs.

Keywords Reactive Executable DSL · Testing · Test Description Language · Debugging · Mutation Analysis

Faezeh Khorram
IMT Atlantique, Nantes Université, École Centrale Nantes
CNRS, LS2N, UMR 6004, F-44000 Nantes, France
E-mail: faezeh.khorram@imt.atlantique.fr
`https://faezeh-kh.github.io/`

Erwan Bousse
IMT Atlantique, Nantes Université, École Centrale Nantes
CNRS, LS2N, UMR 6004, F-44000 Nantes, France
E-mail: erwan.bousse@ls2n.fr
`https://bousse-e.univ-nantes.io/`

Jean-Marie Mottu
IMT Atlantique, Nantes Université, École Centrale Nantes
CNRS, LS2N, UMR 6004, F-44000 Nantes, France
E-mail: jean-marie.mottu@ls2n.fr

Gerson Sunyé
IMT Atlantique, Nantes Université, École Centrale Nantes
CNRS, LS2N, UMR 6004, F-44000 Nantes, France
E-mail: gerson.sunye@ls2n.fr
`https://sunye-g.univ-nantes.io/`

1 Introduction

A wide range of Domain-Specific Languages (DSLs) exist for describing the expected behavior of systems (e.g., [16, 13, 3, 31, 32, 33, 36]). DSLs are used in modeling environments and when the environment offers dynamic Verification and Validation (V&V) techniques, the users (i. e., the domain experts) can also analyze the behavioral models as early as possible to ensure the correctness of their behavior. These techniques require the execution of the models, hence, 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)*.

Testing is probably the most prevalent dynamic V&V technique used for software systems and is commonly enriched with two additional families of techniques: 1) debugging, to localize and fix the defect causing a test case to fail; and 2) test quality measurement, to identify how a test suite can be improved with new test cases. Accordingly, different approaches have emerged in recent years to provide testing support for xDSLs. A first set of approaches aim to provide testing frameworks that are *specific* to selected xDSLs, such as fUML activity diagrams [30, 18] or service-based Business Process Model and Notation (BPMN) processes [26], A second set of approaches aim to provide more generic testing frameworks, i. e. directly compatible with a wide range of xDSLs. Such approaches are much more versatile and can

target large categories of DSLs such as grammar-based compiled DSLs [41] or interpreted xDSLs [20].

However, no testing approach is able to realistically deal with all possible categories of xDSLs. In particular, there are so-called *reactive* xDSLs whose semantics are event-driven, meaning that executed models will react to specific occurring events at runtime. At least two main challenges must be considered to provide testing for reactive xDSLs. First, test cases for conforming models must be described as a scenario of exchanging xDSL-specific events. Hence, the considered testing language for writing test cases must support using those events as test data types. Second, to run such a test case, the execution semantics of the testing language must be somehow connected to the event-driven execution semantics of the reactive xDSL. Thereby, the interaction scenario specified in the test case can be verified by interacting with the tested model during its execution.

In addition, to our knowledge, there is currently no testing approach for xDSLs that is properly integrated with debugging and test quality measurement techniques. For example, while there are interesting interactive debugging approaches for models [5, 6], none is able to support the step-by-step execution of a test case *along with* the step-by-step execution of the tested model. This is especially crucial when testing reactive models that require many interactions, and thus multiple "back-and-forth" between the test case and the model. Likewise, regarding test quality measurement, while a recent approach aims to provide mutation analysis for xDSLs [15], this approach is incomplete as it is not yet able to actually run tests cases on the generated mutants.

Providing such complex facilities—namely testing, debugging, and test quality measurement—for a given *new* reactive xDSL is an expensive and error-prone task. A desirable solution would be a *generic* approach applicable to a wide range of reactive xDSLs. In our previous work [20], we proposed a first generic testing approach for *non*-reactive xDSLs—albeit without any debugging or test quality measurement facilities. Continuing this research direction, the present paper proposes three new core contributions to support testing for reactive xDSLs as well as offering interactive debugging and test quality measurement techniques:

1. Given a reactive xDSL, we enable writing event-driven test cases using the standardized Test Description Language (TDL). We achieve this by automatically generating an event-compatible TDL library for the xDSL, based on the xDSL definition. Then, the domain expert can use xDSL-specific events as test data types—to define both test input data and expected output—when writing TDL test cases.
2. To execute the event-driven test cases on the reactive models, we extend the TDL test execution engine of our previous work [20] by integration with the reactive model execution approach proposed in [24].
3. We offer two test analysis techniques: (i) to diagnose the cause of failure when a test case fails, an interactive debugging facility is provided. It coordinates the initialization and the online interplay of two debugger instances, which are initialized for the test case and its tested xModel from the model debugger proposed in [5]; (ii) to measure the quality of a TDL test suite defined with the proposed approach, we provide an integration of our generic testing approach with the mutation testing framework proposed in [15].

The proposed approach is implemented for the GEMOC Studio, a language and modeling workbench for xDSLs [4]. We applied the approach for two different reactive xDSLs, 'xArduino' i. e., used for modeling Arduino boards and their execution logic, and 'xPSSM' i. e., defined for simulating systems with discrete-event behavior using the Precise Semantics of UML State Machines (PSSM) [36]. Successful use of the approach for writing, executing, and analyzing 247 event-driven test cases for 70 models conforming to two very different reactive xDSLs validates its genericity. Also, to demonstrate the applicability of the test quality measurement feature of the approach, we performed mutation analysis on 65 TDL test suites. They have run on cumulatively 12674 mutants generated for 65 state machines and a mutation score has been calculated for them with success.

Paper organization. Section 2 provides the background and a running example. Section 3 describes an overview of our proposed approach. In Sections 4 and 5, the provided facilities for writing and executing test cases are presented, respectively. Section 6 introduces the provided test analysis techniques. Our tool support is shown in Section 7. In Section 8, the evaluation process and results are illustrated. The related work is presented in Section 9 and the paper concludes in Section 10 with a discussion on future work.

2 Background and Motivation

In this section, we first describe the executable DSLs considered in the scope of this paper. Afterward, we introduce an overview of the Test Description Language (TDL) as well as its adaptation in our previous work for providing testing support for non-reactive xDSLs [20]. To motivate the proposed approach, we also present a running example.

2.1 Executable DSLs (xDSLs)

This paper targets xDSLs composed of at least two parts: an abstract syntax determining the concepts of a particular application domain, and an operational semantics (i. e., the interpreter) defining how the runtime state of a conforming model varies during its execution.

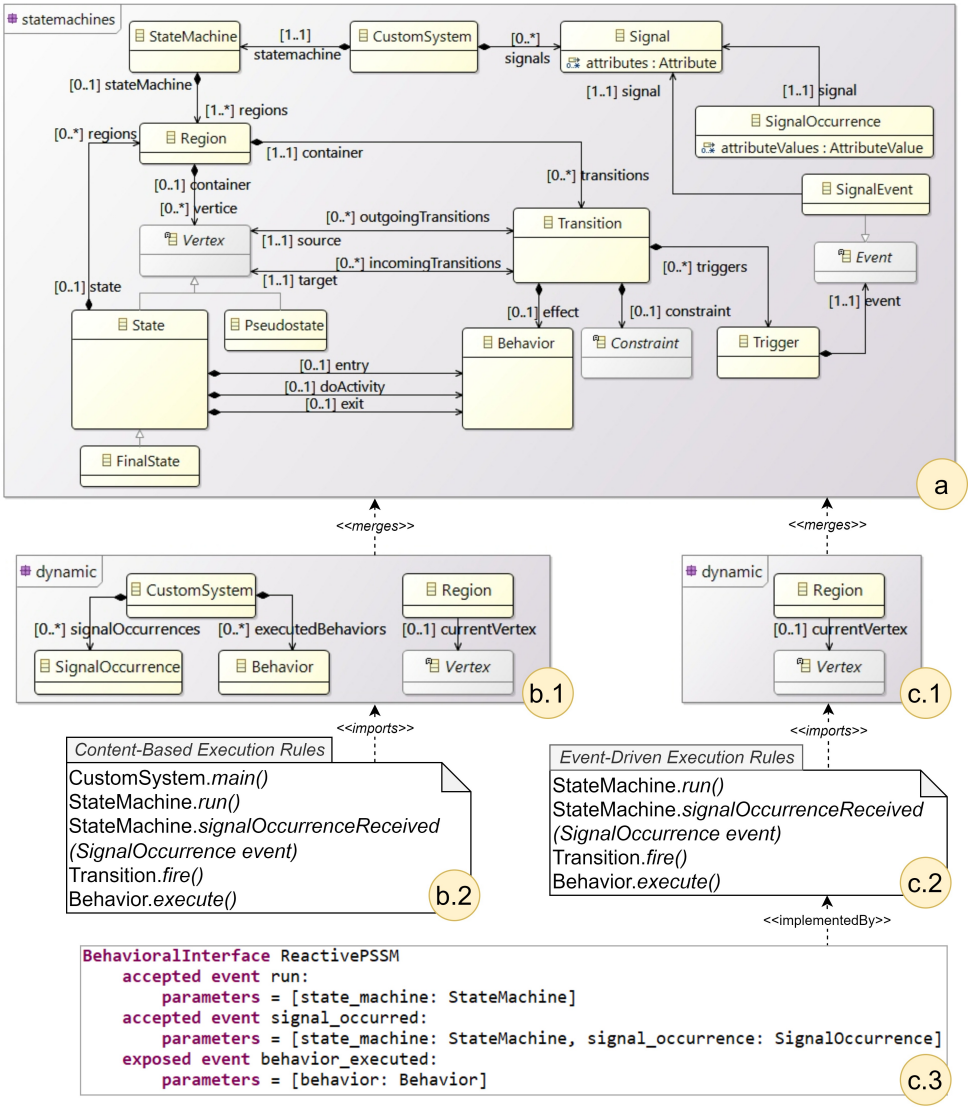


Fig. 1 An xDSL for a subset of UML State Machines conforming to the PSSM specification [36] (referred to as xPSSM). Its semantics is provided in two styles, content-based (b) and event-driven (c).

2.1.1 Running Example: xPSSM

UML State Machines is a well-known subset of the Unified Modeling Language (UML) standard [35] commonly used to model systems with discrete event-driven behavior. The Precise Semantics of UML State Machines (PSSM) is a standardized extension of UML that defines a complete execution semantics for UML State Machines [36]. This paper relies on a simplified version of PSSM as a running example, referred to as xPSSM. xPSSM only contains elements related to the reactive behavior of UML State Machines. Essentially, an xPSSM model is a state machine that can process external occurrences of events and perform behaviors in reaction. Figure 1 shows an overview of different parts of the xPSSM language definition, and we present each part in the reminder of this section.

2.1.2 Abstract Syntax

We consider the abstract syntax of an xDSL to be defined as a metamodel, using a metamodeling language such as MOF [34] or Ecore [40]. Generally, a metamodel is 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.

Part **(a)** of Figure 1 briefly shows the abstract syntax of xPSSM defined as a metamodel. The root element is a CustomSystem. It contains one StateMachine and can have several Signals which will be used in its StateMachine. A StateMachine comprises one or more Region, each represents a behavior fragment that may execute concurrently with other regions if they are owned by either the same State or StateMachine. A Region is a graph comprising a set of

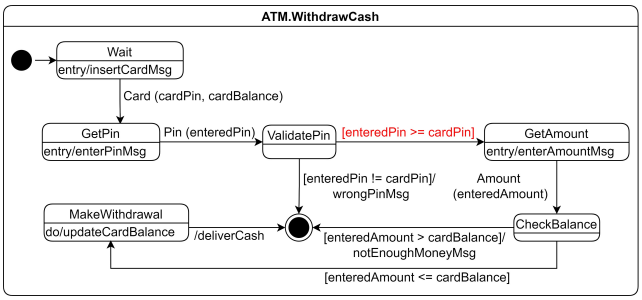


Fig. 2 A sample xPSSM model for cash withdrawal from an ATM. It has a defect since it does not validate the entered pin correctly (the wrong constraint is highlighted in red where $>=$ is used instead of $==$)

Vertices interconnected by Transitions, which determines the behavioral flow within the Region.

Pseudostate and State are two kinds of Vertex. Pseudostates are *transitive*, meaning that the execution passes through them without pause. There are different kinds of Pseudostates such as initial, fork, join, terminate. In contrast, State is a *stable* vertex which means when the execution enters them, it leaves when either some event occurs that triggers a Transition moving to another State or the StateMachine is terminated. A State may have `entry`, `doActivity`, and `exit` Behaviors—in our simplified PSSM, a Behavior is an empty element without any substance. The `entry` and `exit` behaviors are executed when the State is entered and exited, respectively. Execution of the `doActivity` behavior starts after the `entry` Behavior (if any) has completed, and finishes when either it is completed or the State is exited. FinalState is a special kind of State representing the completion of its Region container.

A Transition connects a source vertex to a target one. It can contain three main elements: a Constraint, a Behavior, and several Triggers. A Transition is enabled when its guard Constraint (if any) evaluates to *true*, and its Behavior (if any) is executed once the transition is traversed. The traversal of the transitions may depend on the reception of the event occurrences. This is defined by allocating Trigger elements to them. A Trigger specifies an Event such as SignalEvent whose occurrence (i. e., SignalOccurrence) enables the traversal of the transition containing the Trigger. The SignalOccurrence contains *values* for the *attributes* of its associated Signal. When a state machine receives a signal occurrence, all the enabled transitions that contain a Trigger pointing to the related Signal will be traversed.

Figure 2 shows an example model conforming to xPSSM. It describes a StateMachine that models the behavior of withdrawing cash from an Automated Teller Machine (ATM). The bank `Card`, the entered `Pin` and the `Amount` of withdrawal are Signals whose specific occurrences can be given to the state machine at runtime using SignalOccurrences. The `Card` Signal has two attributes for its pin and balance. This StateMachine has one Region comprising one initial Pseudostate, one FinalState, seven States that three of them have entry Behavior (such as the `insertCardMsg` of the `Wait` State) and one of them has a doActivity Behavior (the `updateCardBalance` of the `MakeWithdrawal` State), and several Transitions which some require signal occurrences to get enabled. For example, the transition from `Wait` to `GetPin` state will be enabled once the state machine receives a SignalOccurrence for the `Card` Signal. Also, the outgoing transition of the `MakeWithdrawal` state has a behavior, namely `deliverCash`.

There are two conditions for a successful withdrawal. First, the entered `Pin` must be equals to the `Card's pin`. It is defined as a Constraint for the outgoing transition of the `ValidatePin` state, but with a wrong operator (highlighted in red in Figure 2). We aim to detect this defect with a test suite written and executed using our proposed approach. Second, the entered `Amount` must be lower than equals to the `Card's balance` (i. e., the Constraint of the outgoing transition of the `CheckBalance` state).

2.1.3 Operational Semantics

The operational semantics of an xDSL must comprise two parts: the definition of the possible runtime states of a model under execution, and a set of execution rules defining how such a runtime state changes over time. We consider the runtime state to be defined in a separate metamodel that introduces new features—later referred to as dynamic features—for the metaclasses of the abstract syntax. This metamodel extends the abstract syntax metamodel using a non-intrusive extension mechanism, such as the UML package merge [35][1]. The execution rules perform an in-place endogenous transformation that defines how the runtime state of a model changes during the execution of the said model.

In this paper, we only consider xDSLs with discrete-event operational semantics (i. e., not continuous). Generally, these semantics can be defined as content-based or event-driven [24]. The former kind executes a model using an initial runtime state for the model that must be provided before the execution starts. The latter kind runs a model through an environment able to interact with the model execution through event occurrences. In the following, we clarify their differences as well as their requirements for testing support.

Content-Based Semantics A content-based semantics defines how to run a model in a *closed* environment, where only an initial runtime state is provided to the model before it is started. The execution rules of a content-based semantics comprise at least one rule acting as a starting point for the model execution, usually called the main(). This rule can trigger other execution rules (if any), and each may call

[1] There are also other ways to define the runtime state, such as using imports or inheritance relationships.

other rules and perform observable execution steps in order to finalize the execution.

For example, we defined a content-based semantics for xPSSM. First, the runtime state definition is shown in part **(b.1)** of Figure 1. The `currentVertex` is a dynamic feature of the Region, which is used to remember the last executed Vertex at each execution step. The `signalOccurrences` dynamic feature holds an ordered list of the signal occurrences that should be dispatched to a state machine, i. e. the required input for the state machine execution. The `executedBehaviors` keep track of the Behavior instances of the state machine that have been executed. It is required for testing purposes and will be used in Section 2.3. We also defined a set of content-based execution rules (**(b.2)** part of Figure 1) that are explained below using the running example.

In Figure 2, the *ATM.WithdrawCash* state machine relies on three different signals for its execution, labeled `Card`, `Pin`, and `Amount`. The content-based semantics of xPSSM requires that the sequence of concrete occurrences of said signals be prepared before the execution and stored all at once in the `signalOccurrences` dynamic feature. The `main()` rule then starts the execution which results in activating the `Wait` state and executing its `insertCardMsg` entry behavior, hence adding this behavior to the list of `executedBehaviors`. Afterwards, the `main()` rule calls the signalOccurrenceReceived(event) rule on each of the provided signal occurrences in order. Here, first the `Card` occurrence will be dispatched which enables the transition to the `GetPin` state. Then, the `Pin` occurrence will be executed, hence traversing the transition to the `ValidatePin` state. The execution continues until either a signal occurrence is required (e. g., after entering the `GetAmount` state, an occurrence for the `Amount` signal is required to exit) or the execution reaches a FinalState (e. g., in the `ValidatePin` state, if the entered pin is wrong, the transition to the FinalState will be traversed and the execution will be terminated).

Event-Driven Semantics Although it is possible to execute a model solely based on its content, there are many cases requiring dynamically interacting with a running model, e. g. for running a co-simulation with other models [6, 24]. This requires the xDSL's operational semantics to have a real *event-driven* behavior that precisely specifies how one can interact with a running model, and how the said model should react. In this paper, we consider that this aspect is handled by a language component called a behavioral interface, which we introduce in the next section as the foundation for the *event-driven semantics* of an xDSL.

2.1.4 Behavioral Interface

The behavioral interface of an xDSL specifies the types of events that can be sent to and received from conforming models during their execution. While different approaches can be used to define such an interface (e. g., [7], [24]), this paper uses the metalanguage proposed in [24]. This metalanguage specifies that a behavioral interface comprises a set of accepted and exposed events, each containing parameters. An accepted event specifies what can be accepted by a running model and an exposed event determines its observable reactions. While this metalanguage allows events to be processed asynchronously by the model (i. e. the model can receive new event occurrences[2] while still in the middle of processing one), in the present paper we only consider events that are processed synchronously—often referred to as a *run-to-completion* semantics.

For example, the **(c.3)** part of Figure 1 shows a behavioral interface for xPSSM containing three event definitions:

- accepted event *run*: triggers the initialization of its state machine parameter.
- accepted event *signal_occurred*: takes a signal occurrence as parameter and triggers its corresponding execution steps in the state machine.
- exposed event *behavior_executed*: notifies the execution of the Behavior elements.

The behavioral interface of an xDSL must be implemented by the execution rules of its operational semantics. For instance, the **(c.1)** and the **(c.2)** parts of Figure 1 present an event-driven semantics for xPSSM (the runtime state definition and the execution rules, respectively). The run() execution rule implements the accepted event run, the signalOccurrenceReceived(event) rule implements the accepted event signal_occurred, and the execute() rule implements the exposed event behavior_executed.

As an example, to execute the *ATM.WithdrawCash* state machine (Figure 2), event occurrences conforming to xPSSM's behavioral interface should be communicated to the state machine. One can first send a run event with the *ATM.WithdrawCash* state machine as its parameter. This starts the execution and resulted in activating the `Wait` state and executing its `insertCardMsg` entry behavior which will be exposed by the model through a behavior_executed event. It is indeed the state machine reaction to receiving the run event occurrence. As the `currentVertex` is the `Wait` state, an occurrence for the signal_occurred event must be sent to the state machine with a `Card` instance to pursue.

Therefore, unlike content-based semantics, event occurrences are given to a running model one by one, who then performs observable reactions at runtime. Consequently, it is possible to send different event occurrences to a model based on the responses that it provides, i. e. to dynamically react to the model's observable actions. This is especially useful for techniques that benefit from dynamic interactions

[2] Please note that the "occurrence" word is used in the paper in two ways, for reactive xDSLs in general, and for xPSSM in particular.

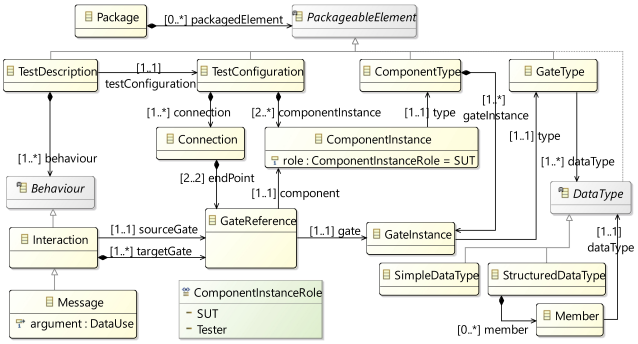


Fig. 3 An Excerpt of the TDL Metamodel [10]

with a model, such as interactive debuggers, testing frameworks, or co-simulation environments.

In the remainder of the paper, xDSLs with content-based semantics are called *non-reactive* xDSL, while xDSLs with event-driven semantics are called *reactive* xDSL.

2.2 The Test Description Language (TDL)

The Test Description Language (TDL) was introduced by the European Telecommunications Standards Institute (ETSI) as a generic language for describing test cases. TDL supports describing test objectives derived from system requirements and defining test cases that refine those objectives [27]. The standard semantics of TDL provides a loose semantics written in natural language [10] and a precise translational semantics using the Testing and Test Control Notation version 3 (TTCN-3) as a target language [11]—TTCN-3 is also standardized by the ETSI. A reference implementation of TDL is also provided, containing a standard abstract syntax, textual and graphical concrete syntax, and tools for model validation, and transformation to TTCN-3, among others.

Figure 3 shows the main elements of the TDL abstract syntax. A Package is the root element of a TDL model, hence the container of all other elements. To define a complete test case, three main information are required:

Test Data: The first step in defining test data is to determine the required data types. TDL does not provide any concrete data type since its main objective is to be generic and platform-independent. So the testers should define their required types using the DataType element, then instantiating them to define test data, both the input data that will be sent to the System Under Test (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).

Test Configuration: A test configuration specifies a communication protocol between the test suite (later referred to as the *test system*) and the SUT. TDL follows a component-based approach, hence a TestConfiguration comprises two or more ComponentInstances, one in the role of SUT and the rest as Tester as well as the Connections between the components. A ComponentInstance is typed by a ComponentType, which determines the component communication channels using the so-called gates. Accordingly, it contains at least one gate (i. e., GateInstance) that is instantiated from a GateType. A GateType defines what kind of data can be exchanged through its instances.

Test Description: To describe the behavior of a test case, the TestDescription element should be instantiated. It uses one of the previously defined TestConfiguration instances, and contains a sequence of Behavior elements. Currently, twenty types of behavior are defined in the TDL standard, such as Message, TimeOut, AlternativeBehavior, etc. Examples of TDL test cases are given shortly after.

2.3 TDL-based Testing Support for Non-Reactive xDSLs

In our previous work [20], we proposed a generic testing approach for non-reactive xDSLs using TDL. Two roles were involved in the approach: a language engineer who implements a non-reactive xDSL according to the definitions given in Section 2.1, and a domain expert who uses this xDSL to define behavioral models and wishes to write test cases for them. Our proposed approach provided all the required material for the domain experts to write and execute TDL test cases for their models. Its main components which are used and extended in this paper are described in Sections 4 and 5. For more information, we refer the reader to the paper [20].

For example, this approach can be used to write TDL test cases for xPSSM models when they are executed by the xPSSM content-based semantics (part **(b)** of Figure 1). Figure 4 depicts an excerpt of a TDL test case for the *ATM.WithdrawCash* State machine. Each arrow corresponds to a Message TDL Behavior that carries some data and is exchanged between the `Test Component` and the `SUT`. When the sender of a Message is the `Test Component`, the ex-

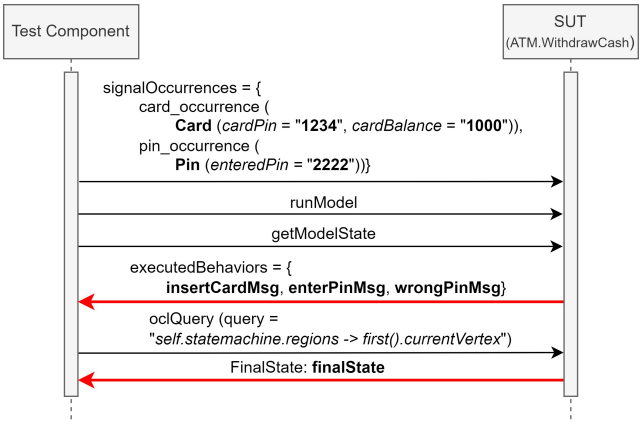


Fig. 4 A TDL test case for the running example written using our previous work [20]

changed data is test input data or a request, otherwise, it is some expected output (i. e. an assertion).

Our approach allows the domain expert to use the domain concepts when defining test data. For instance, in the first Message in Figure 4, two signal occurrences, one for the Card and one for the Pin are defined and sent to the state machine. This puts the state machine in an initial runtime state as this data will be set as the value of its signalOccurrences dynamic feature. We intentionally use a wrong pin number to see whether the test case can detect the defect of the state machine. Then, the test component requests to run the model (runModel) and get its runtime state after execution (getModelState). These two operations are also provided by the approach. To define the expected output, the domain expert can again use the domain concepts, in this case the executedBehaviors dynamic feature with its expected value i. e., three Behavior instances of the state machine including `insertCardMsg`, `enterPinMsg`, and `wrongPinMsg`. As shown in Figure 4, the assertion is failed (the first red arrow) because due to the defect of the model, the value of the executedBehaviors is {`insertCardMsg`, `enterPinMsg`, `enterAmountMsg`} It means the test case detects the defect of the model successfully.

Besides, the approach offers facilities to use Object Constraint Language (OCL) queries in the TDL test cases. For instance, in the test case of Figure 4, the test component sends an OCL query to the model to retrieve the value of its currentVertex feature. The expected result is finalState but it is failed since the model is wrong and at this point, it is in the GetAmount state.

2.4 Motivation and Requirements

Our previous work [20] provides testing support for an xDSL under two conditions: (1) it is possible to provide input data to the model under test when initializing its runtime state, (2) it is possible to get output data at the end of the execution by retrieving the final runtime state of the model, which can then be compared with some expected state. For example, in the test case shown in Figure4, input data is provided by giving an initial value to the `signalOccurrences` dynamic feature, and the `executedBehaviors` dynamic feature is retrieved to be compared with some expected sequence.

However, given a reactive xDSL, a running model should only communicate data using the behavioral interface of the xDSL. Accordingly, a prospective test case for a reactive model should be described as a scenario in which the test system sends events to the model and checks whether the model sends back the expected event. For instance, if we write a test case for the ATM state machine, the test component shall send a *signal_occurred* event with a signal occurrence for the Card signal to the running state machine, and check whether it reacts by exposing a *behavior_executed* event for the `enterPinMsg` Behavior. Therefore, test cases of reactive models must be written differently, hence leading to the following first requirement:

Req.1 The testing language used for writing test cases should allow the domain expert to use the events specified in the xDSL's behavioral interface as test data types.

Then, running tests on a model obviously requires a way to execute the model, which can be performed differently depending on how the xDSL's semantics is defined. For non-reactive xDSLs, the model execution is a one-time operation, while for reactive xDSLs, it is driven by exchanging events at runtime. This means for executing a test case on a reactive model, the model should keep running and interacting with the test case until the test case is terminated; is passed, failed, or has an inconclusive result. Thus for providing test execution support, there is a second requirement:

Req.2 The test runner should be able to run event-driven test cases by online interaction with an event-driven model execution engine that can run reactive models.

As complex interactions may occur between the test cases and the models, it is difficult to diagnose the point of failure when a test case fails. In such cases, a synchronized interactive debugging facility would be greatly helpful and would allow (1) to execute and observe step-by-step the test case and its tested model both at the same time; and (2) to jump from one execution to another when an interaction occurs between them. Therefore, we consider this third requirement for the proposed approach:

Req.3 An interactive debugging facility is required for diagnosing the cause of failure in the failed test cases.

With these three requirements fulfilled, testers will be able to run their tests and debug failed ones. However, if test cases do not find any bug, while it may validate the correctness of the tested model (in the best case), it may also highlight weaknesses of the test suite (in the worst case). To properly measure the quality of a test suite, one well-known efficient technique is mutation analysis [19], which can produce a score representing the overall quality of the test cases. We therefore decided to consider the following final requirement for the proposed approach:

Req.4 The approach should support test suite quality measurement based on mutation analysis.

In the next sections, we present our approach providing testing and debugging facilities for reactive xDSLs on the basis of our previous work. We fulfill all the aforementioned requirements with, respectively, generating an event-compatible TDL library for a given reactive xDSL to fulfill **Req.1**, integrating the TDL interpreter with event-driven model execution tools to fulfill **Req.2**, adapting interactive debugging facilities for TDL test failure diagnosis to fulfill **Req.3**, and providing a TDL test quality measurement tool based on mutation analysis to fulfill **Req.4**.

3 Approach Overview

Figure 5 presents an overview of the proposed approach[3]. At the top left corner, we assume that a language engineer has implemented a reactive xDSL based on the definitions given in Section 2.1. The domain expert on the right defines a system with reactive behavior by instantiating models from the provided xDSL. She/he wishes to test and debug those models to ensure they behave as expected.

The *TDL Library Generator* is the first component of the approach (at the top center) that was initially introduced in our previous work [20]. Its first version produced a *domain-specific TDL library* for a given non-reactive xDSL, providing all the data types required for the specification of test data, a set of default test configurations, and elements for requesting the execution of the tested models and of OCL queries [20]. In this paper, we extend this component to support reactive xDSLs. Through this extension, the generated library also provides an event-compatible TDL package generated from the definition of the xDSL's behavioral interface. This package provides the required elements for writing and executing event-driven TDL test cases for reactive models. Details are given in Section 4.

Executing TDL test cases on the models is the role of the *TDL Interpreter* component (at the center) that we initially proposed in [20]. As Figure 5 shows, it has connections with three external components: the *Execution Engine*, the *Query Evaluator*, and the *Event Manager*. The first two connections are from our previous work, enabling performing operations on the 'non-reactive' tested models and running OCL queries on them, respectively. This paper extends the interpreter with a new connection to an *Event Manager* to provide execution of event-driven TDL test cases on the 'reactive' models. We assume that an *Event Manager* exists which provides services to interact with a running reactive model. More precisely, given a reactive xDSL, it enables the external tools such as testing tools to exchange events conforming to the xDSL's behavioral interface with the models conforming to the xDSL's abstract syntax at runtime. In Section 5, this component is explained in more detail.

[3] Elements of the Figure are written in italic in the text.

Finally, the approach offers two test analysis techniques for the domain expert. First, *Interactive Debugging* (at the top center) to help the domain expert to find out the cause of a failure in a test case. It can be used to debug interactively the test case and its model under test at the same time, so the domain expert can observe gradually the model's reaction to the reception of requests from the test case. Second, *Mutation Analysis* (at the bottom center) to help the domain experts to measure the quality of their written TDL test cases. In a nutshell, given a TDL test suite for a model, it performs mutation analysis on the model and calculates a mutation score for the test suite which can be used for measuring its quality. These analysis techniques support both non-reactive and reactive xDSLs and are presented in Section 6.

4 Support for Writing Event-Driven TDL Test Cases

This section presents how the approach provides facilities for writing test cases for reactive models using TDL. At first, we describe what should an event-driven test case look like through an example. Then we introduce the *TDL Library Generator* which enables writing such test cases in TDL by providing a TDL library specific to a given reactive xDSL. Finally, we show how the domain expert can use the library to write executable test cases for reactive models.

4.1 A Sample Event-Driven Test Case

We mentioned in Section 2.4 that an event-driven test case for a reactive model should be described as a scenario of exchanging events between the test system and the tested model. Figure 6 shows such a test case for the *ATM.WithdrawCash* state machine (previously shown in Figure 2). The test case aims to check that the ATM does not accept an incorrect pin code, so it must be able to uncover the defect of the model. As can be seen, the events used in the test case conform to the xPSSM's behavioral interface (the **(c.3)** part of Figure 1) and their parameters are references to the elements of the ATM state machine.

First, the test component sends a *run* event to request the start of the execution and expects to receive in return a *behavior_executed* event for the `insertCardMsg` behavior. This assertion passes (the first green arrow in Figure 6) because according to Figure 2, when the state machine initializes, the execution should enter the `Wait` state, execute its `entry` behavior named `insertCardMsg`, and wait there until one of its outgoing transitions can be traversed.

Next, the test component sends a *signal_occurred* event with an occurrence of the `Card` signal and expects to receive in return a *behavior_executed* event for the `enterPinMsg` behavior. As the state machine execution is currently in

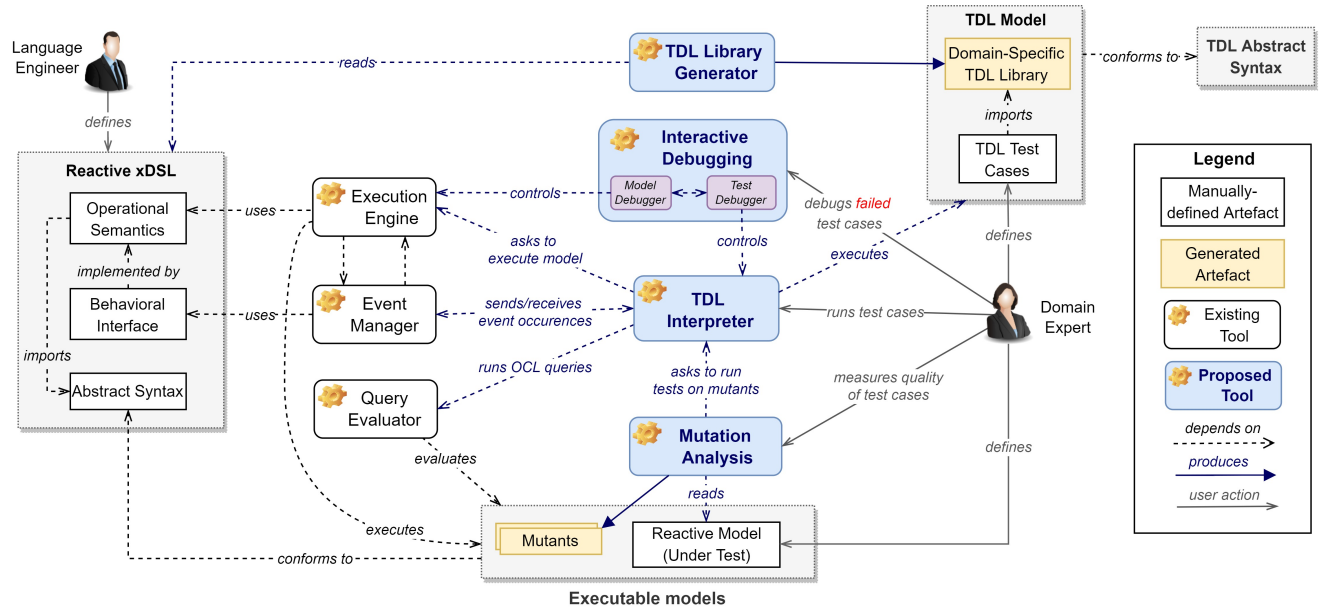


Fig. 5 Overview of the Proposed Approach

the `Wait` state, by receiving this event from the test component, the transition to the `GetPin` state will be traversed. So the execution enters this state and runs its `entry` behavior named `enterPinMsg`. Therefore, the second assertion also succeeds (the second green arrow in Figure 6).

Afterwards, the test component sends another *signal_occurred* event with an occurrence of the `Pin` signal and since the value of the entered pin (i. e., 2222) is different from the the card's pin (i. e., 1234), it expects to receive a *behavior_executed* event for the `wrongPinMsg` behavior. According to Figure 2, as the state machine execution is currently in the `GetPin` state, receiving this event from the test component resulted in traversing the transition to the `ValidatePin` state. At this point, the constraints of its outgoing transitions are evaluated to check if they are enabled. However, as explained earlier, the ATM state machine contains a defect: an equality sign was mistakenly replaced by a superior-or-equal sign, leading to the wrong constraint "enteredPin >= cardPin". Consequently, instead of enabling the transition to the `finalState`, the one to the `GetAmount` state is enabled. Therefore, the `wrongPinMsg` event is never observed, meaning that the third assertion of the test case fails (the first red arrow in Figure 6).

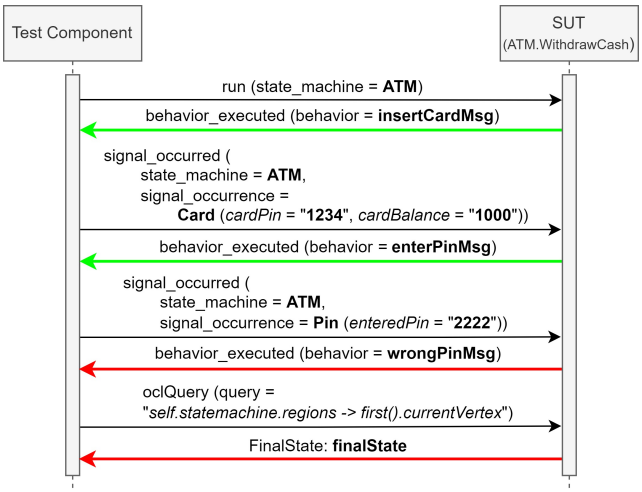


Fig. 6 A potential event-driven TDL test case for the running example, with two passed and two failed assertions.

Finally, the test component sends an OCL query to check whether the `currentVertex` is the `finalState`. As described above, due to the defect of the model, the execution is currently in the `GetAmount` state, so the assertion fails (the second red arrow in Figure 6). In the remainder of the section, we explain how our proposed approach provides facilities for the domain expert to write such event-driven test cases for any reactive model.

4.2 TDL Library Generator for Reactive xDSLs

The main objective of the *TDL Library Generator* component is to provide a domain-specific TDL library for a given xDSL. Such a library aims to provide a set of TDL elements for the domain experts, allowing them to write TDL test cases for the models conforming to the considered xDSL. As shown in Figure 5, the generator reads the definition of an xDSL—in particular, the abstract syntax, the parts of the operational semantics defining the possible runtime states of the conforming models, and the behavioral interface—and produces a TDL library specific to the xDSL which contains four TDL packages:

1. **xDSL-Specific Types Package**, containing all the TDL data types required for the specification of test data.
2. **Common Package**, providing TDL elements common to any given xDSL, including a set of elements for performing operations on the model under test and elements for enabling the use of OCL queries in the test cases.
3. **xDSL-Specific Events Package**, with the TDL definition of the events of the xDSL's behavioral interface.
4. **Test Configuration Package**, providing a default test configuration to be used by the TDL test cases written for executable models.

Packages 1,2,4 are generated for any xDSL (i. e., either non-reactive or reactive) and their generation is already mostly explained in our previous work [20]. However, Package 3 (the *xDSL-Specific Events Package*) is a new package generated solely for reactive xDSLs. In what follows, we present in order how each package is generated. For Packages 1 and 2, we summarize how they can be generated using our previous work [20]. For Package 3, we describe our novel generation process based on the events defined in a given behavioral interface. For Package 4, we explain how we upgraded the generator of our previous work [20] to obtain test configurations adapted for reactive xDSLs. Afterwards, we present how these packages can be used for writing test cases for reactive models.

4.2.1 Generation of the xDSL-Specific Types Package

As discussed in Section 2.2, test data is composed of instances of TDL data types. Therefore, to use TDL for a specific domain, all the required data types must be defined beforehand. To avoid having the tester manually creating these types, we automated this task in our previous work by proposing a model transformation from Ecore to TDL. This transformation generates TDL data types for a given xDSL from its definition, mainly the abstract syntax and the definition of the runtime state. Listing 1 shows some of the generated TDL data types for the xPSSM Ecore metamodel (the (**a**) part of Figure 1), including CustomSystem, Signal, SignaOccurrence, StateMachine, and Behavior. Using this generated package, the domain expert can easily define model elements in TDL and use them as test data.

```
Package xPSSMTypes {
  Type CustomSystem (
    statemachine of type StateMachine,
    signals of type Signal);
  Type Signal ( attributes of type Attribute) ;
  Type SignalOccurrence (
    signal of type Signal,
    attributeValues of type AttributeValue);
  Type StateMachine (
    _name of type EString, regions of type Region);
  Type Behavior(_name of type EString);
  ...
}
```

Listing 1 Some of the TDL Data Types generated for the xPSSM DSL

4.2.2 Generation of the Common Package

The common package comprises a set of elements referred to as *model execution commands* for performing several operations on the model under test, such as `runModel` for requesting its content-based execution, `resetModel` for resetting its state to the default, and `getModelState` for getting its current state, i. e., the content of its dynamic features. In addition, it provides a TDL element as `oclQuery (query = ?)` that lets the test component to send OCL queries to the tested model by setting the value of its `query` argument. Accordingly, the result of the query evaluation can be used when defining a test oracle.

4.2.3 Generation of the xDSL-Specific Events Package

As discussed in Section 2.4, the first requirement for a generic testing approach for reactive xDSLs is to allow the domain expert to use events as test data types when writing test cases. This means for a given reactive xDSL, the testing language of the approach should support using the events of the xDSL's behavioral interface in the test cases. Since our approach uses the TDL testing language, we need the definition of the events in TDL, which we provide by the *xDSL-Specific Events* package.

This package is automatically generated by a transformation from the behavioral interface metalanguage [24] to TDL. Table 1 shows the outline of the transformation rules. In a nutshell, a `BehavioralInterface` is transformed to a TDL `Package` that is the container of other elements. Each `Event` is transformed to a `StructuredDataType` which is annotated according to the `EventType` and comprises `Members` generated for the `EventParameters`. To assign the type of `Members`, the content of the previously generated *xDSL-Specific Types Package* is used.

BI element	Generated TDL element
`Behavioral-Interface`	`Package` containing all the other generated elements
-	`Import` the xDSL-Specific Types `Package`
`EventType`	`Annotation`
`Event`	`StructuredDataType` containing one `Member` per `Parameter` and annotated based on it type
`Event-Parameter`	`Member`. Its type is set using the TDL `DataTypes` provided by the imported `Package`

Table 1 Behavioral Interface to TDL Transformation Rules

```
1  Package xPSSMEvents {
2    Import all from xPSSMTypes;
3
4    Annotation AcceptedEvent;
5    Annotation ExposedEvent;
6
7    Type run (state_machine of type StateMachine) with
8      {AcceptedEvent;};
9    Type signal_occurred (
10       state_machine of type StateMachine,
11       signal_occurrence of type SignalOccurrence
12   ) with {AcceptedEvent;};
13   Type behavior_executed (behavior of type Behavior) with
14     {ExposedEvent;};
15 }
```

Listing 2 TDL elements generated for the xPSSM behavioral interface

Listing 2 shows the *xDSL-Specific Events Package* generated for the xPSSM's behavioral interface. To distinguish *accepted events* from *exposed events*, two Annotation elements are generated (lines 4-5). For each event of the xPSSM's behavioral interface (part **(c.3)** in Figure 1), a TDL Type is produced (lines 7-14) and is annotated with one of the Annotation elements according to the type of the event. For example, the TDL Type generated for the *run* event is annotated as `AcceptedEvent` (line 8), and the one for *behavior_executed* event is annotated as `ExposedEvent` (line 14).

The parameters of the events are transformed in Members of the TDL Types. For example, line 7 shows the Member generated for the *state_machine* parameter of the *run* event. Since the parameters are references to the model elements, their type conforms to the xDSL's abstract syntax. Thanks to the generated *xDSL_Specific types package*, we have the definition of all the required data types in TDL. Therefore, we can use them to assign the type of the Members. For instance in Listing 2, the `xPSSMTypes` package is imported (line 2) and its content i. e., the TDL Types generated for the xPSSM metamodel is used several times (e. g., the `StateMachine` in line 7 or the `Behavior` in line 13).

4.2.4 Generation of the Test Configuration Package

Although the *xDSL-Specific Events Package* provides the required elements for writing event-driven TDL test cases, we need to define how the test system can get connected to the reactive model under test to run such test cases. This information can be expressed using TDL Test Configuration elements. In particular, a TDL test configuration defines what are the available communication gates, each gate allowing specific types of requests. In our previous work for non-reactive xDSLs, we considered that a test case exchanges only two kinds of messages with the model under test: either model execution commands or OCL queries. In the present work, we add a new kind of requests which correspond to all events of the behavioral interface of the considered xDSL.

```
1  Package testConfiguration {
2    Import all from common;
3    Import all from xPSSMEvents;
4
5    Gate Type genericGateType accepts modelExecutionCommand;
6    Gate Type oclGateType accepts OCL;
7    Gate Type reactiveGateType accepts run , signal_occurred ,
       behavior_executed;
8    Component Type component having {
9      gate genericGate of type genericGateType;
10     gate oclGate of type oclGateType;
11     gate reactiveGate of type reactiveGateType;
12   }
13   Annotation MUTPath;
14   Annotation DSLName;
15
16   Test Configuration xPSSMConfiguration {
17     create Tester tester of type component;
18     create SUT statemachine of type component with {
19       MUTPath: 'TODO : Put the path to the MUT';
20       DSLName: 'org.imt.pssm.reactive.ReactivePSSM';
21     };
22     connect tester.genericGate to statemachine.genericGate;
23     connect tester.oclGate to statemachine.oclGate;
24     connect tester.reactiveGate to statemachine.reactiveGate;
25   }
26 }
```

Listing 3 TDL test configuration package generated for the xPSSM DSL

Listing 3 shows an example of *Test Configuration Package* generated for the xPSSM DSL. It has three Gate Types: the first two are defined for exchanging `modelExecutionCommands` (line 5) and `OCL` queries (line 6) provided by the `common` Package (imported in line 2), and the third is added in this paper to communicate events (line 7) provided by the `xPSSMEvents` Package (imported in line 3). There is also a Component Type comprising one gate instance for each Gate Type (lines 8-12). Finally, a Test Configuration is defined containing two Component Instances, one of the Tester kind (line 17) and one of the SUT kind (lines 18-21). The SUT requires information about the model under test, including the path to the model (the `MUTPath` annotation in line 19) that should be set by the domain expert, and the name of the DSL that the model conforms to (the `DSLName` annotation in line 20) which is automatically set by the TDL Library Generator. The test configuration also specifies how

```
1  Package reactiveATM_testSuite {
2    Import all from common;
3    Import all from xPSSMTypes;
4    Import all from xPSSMEvents;
5    Import all from testConfiguration;
6
7    StateMachine ATM (_name = "withdrawCash");
8    Behavior insertCardMsg (_name = "insertCardMsg");
9
10   Test Description test_wrongPin uses configuration xPSSMConfiguration{
11     tester.reactiveGate sends run (state_machine = ATM) to statemachine.reactiveGate;
12     statemachine.reactiveGate sends behavior_executed (behavior = insertCardMsg) to tester.reactiveGate;
13     tester.reactiveGate sends signal_occurred (state_machine = ATM,
14       signal_occurrence = card_occurrence ( signal = Card,
15         attributeValues = {cardPinValue (value = "1234"), cardBalanceValue (value = "1000")})
16       ) to statemachine.reactiveGate;
17     statemachine.reactiveGate sends behavior_executed (behavior = enterPinMsg) to tester.reactiveGate;
18     tester.reactiveGate sends signal_occurred (state_machine = ATM,
19       signal_occurrence = pin_occurrence ( signal = Pin, attributeValues = {enteredPinValue (value = "2222")})
20       ) to statemachine.reactiveGate;
21     statemachine.reactiveGate sends behavior_executed (behavior = wrongPinMsg) to tester.reactiveGate;
22     tester.oclGate sends oclQuery (query =  "self.statemachine.regions->first().currentVertex") to statemachine.oclGate;
23     statemachine.oclGate sends finalState to tester.oclGate;
24   }
25 }
```

Listing 4 An event-driven TDL test case for testing the running example

the test system connects to the SUT through the definition of the Connections between their Gate instances (lines 22-24).

4.3 Using the TDL Library to write Event-Driven Tests

In Section 4.1, we described an overview of an event-driven test case (Figure 6) for the running example (Figure 2). By using the TDL Library generated for the xPSSM DSL, the domain expert can write such a test case in TDL that will be executable. It is presented in lines 10-24 of Listing 4. Using the data types provided by the `xPSSMTypes` package (imported in line 3), the domain expert can define model elements to use them as test data, such as using StateMachine and Behavior data types to define the `ATM` and the `insertCardMsg` elements, respectively. Note that, we do not present all the defined test data in Listing 4, but the complete TDL code is accessible on a public GitLab server.

The test case uses the `xPSSMConfiguration` (line 10) provided by the `testConfiguration` package (imported in line 5) and is defined as a sequence of exchanging data and/or requests between the gates of the `Tester` and `SUT` component instances. When the data is an event, it should be exchanged through the `reactiveGate` of the components (lines 11-21), and when the data is either an OCL query or an expected output related to the query evaluation result, the `oclGate` should be used (lines 22-23).

By importing the generated `xPSSMEvents` package (line 4), the domain expert defines event instances and then uses them as test data. For example, in line 11, the `tester` sends a `run` event for the `ATM` state machine to the model under test, so the event is used as test input data. Afterwards, an assertion is defined where the expected output is a `behavior_executed` event for the `insertCardMsg` behavior (line 12), so the event is used as expected output.

5 Support for Executing Event-Driven TDL Test Cases

In this section, we present the *TDL Interpreter* component which is responsible for executing TDL test cases on models. First, we describe its required external components and then we explain its test execution algorithm.

5.1 Required External Components

As illustrated in Figure 5, the TDL Interpreter needs connections with three external components. We assume they already exist and provide services as follows:

- **Execution Engine**: provides services to manage the execution of the models such as running the model, resetting its state to default, and getting its current state. This component uses the operational semantics of an xDSL to execute its conforming models.
- **Query Evaluator**: can trigger the evaluation of an OCL query on a model and retrieves the result.
- **Event Manager**: provides services to send event occurrences to a running reactive model and to receive event occurrences exposed by the model. As running the model is performed by an execution engine, this component is also connected to the execution engine to communicate event occurrences with running models.

The first two connections were presented in detail in our previous work. The TDL Interpreter is connected to the execution engine to interpret the *model execution commands* used in a TDL test case, and is connected to the query evaluator to interpret the *OCL queries* written in a TDL test case and to use the query evaluation result when required by a test oracle. These two connections enabled our approach to run TDL test cases on 'non-reactive' models [20]. In this paper, we add the third connection which is necessary for executing event-driven TDL test cases on 'reactive' models.

5.1.1 Connection to Event Manager

As already explained in Section 4, the *TDL Library Generator* provides the TDL definition for the events of an xDSL's behavioral interface (i. e., the generated *xDSL-Specific Events Package*) along with the required TDL gates for exchanging them between the test system and the model under test (i. e., the *reactiveGate* in the *Test Configuration Package*). Accordingly, we extended the *TDL Interpreter* to be able to interpret these new elements, hence executing the event-driven TDL test cases. To this end, we introduce a new integration for the TDL Interpreter with an external component called *Event Manager*. The Event Manager must be configurable for a given reactive xDSL and allow external tools (e. g., testing tools) to interact with the xDSL's conforming models based on the xDSL's behavioral interface using two services: sending accepted event occurrences to a running model, and receiving its observable reactions as occurrences of the exposed events.

5.1.2 Overall Architecture

The UML class diagram presented in Figure 7 shows the overall architecture of the TDL interpreter. As we mentioned earlier, an execution engine uses the operational semantics of an xDSL to execute its conforming models and an event manager uses the behavioral interface of an xDSL and is connected to an execution engine. To implement the operational semantics and the behavioral interface, different

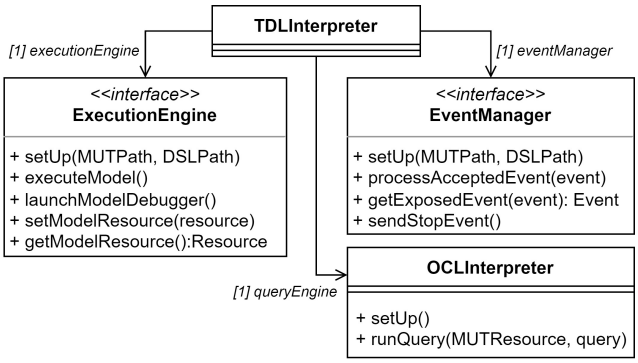


Fig. 7 Class diagram showing the associations of the TDL Interpreter

metaprogramming approaches—i. e. one or several metalanguages used in a particular fashion— can be used. Consequently, various execution engines and event managers may exist, each supporting a specific metaprogramming approach. To make the TDL Interpreter agnostic to this heterogeneity, we defined its required interfaces.

The ExecutionEngine interface is mostly similar to the one from our previous work [20]. It can be used for setting up the execution engine based on the model under test and its conforming xDSL, for executing the model, for setting the model in a specific runtime state, and for getting its current state. As this paper proposes interactive debugging facilities, we added a new method to this interface to launch a debugger instance for the model under test. More details on these debugging facilities are provided in Section 6.

This paper introduces an interface for the EventManager, comprising methods for setting up for a specific model and its conforming reactive xDSL, accepting an event to process on the model, retrieving an expected exposed event from the events exposed by the model, and stopping the communication with the model and releasing the resources. For the OCL Query Evaluator, we also rely on a specific interface named OCLInterpreter from our previous work [20].

5.2 Test Execution Algorithm of the TDL Interpreter

In this section, we provide the details of the TDL Interpreter definition, mainly its test execution algorithm. Please note that all algorithms presented here are upgraded versions of the algorithms originally presented as part of our previous work [20] and are extended here to support running tests on reactive models using an *Event Manager*. Algorithm 1 shows the main loop, which requires as input a TDL package containing the set of TDL test cases to execute. For each test case, its test configuration must be activated first (line 3) using Algorithm 2. As can be seen, the path to the model under test and the name of the DSL are first retrieved from the annotations of the SUT component. Then, based on the connections between the gates, the required external com-

Algorithm 1: The TDL Interpreter main loop

```
   Input:
   package: the TDL package containing the TDL test cases to
   be executed
1  begin
2      foreach testcase ∈ package.testCases do
3          testcase.configuration.activate()
4          foreach behavior ∈ testcase.behaviors do
5              if behavior is Message then
6                  sourceGate ← behavior.source
7                  targetGate ← behavior.target
8                  if sourceGate.component.role is Tester
                     then
9                      request ← behavior.argument
10                     targetGate.sendRequestToSUT(request)
11                 else if sourceGate.component.role is SUT
                     then
12                     testOracle ← behavior.argument
13                     targetGate.assert(testOracle)
14             else if behavior is <other behavior types> then
15                 ...
```

Algorithm 2: Activating test case configuration

```
   Input:
   configuration: TDL test configuration to be activated
1  begin
2      MUTPath ← configuration.sutComponent.MUTPath
       DSLName ← configuration.sutComponent.DSLName
       foreach connection ∈ configuration.connections do
3          if connection between generic gates then
4              engine ←new ExecutionEngine()
5              engine.setUp(MUTPath, DSLName)
6          if connection between OCL gates then
7              OCLInterpreter ← new OCLInterpreter()
8              OCLInterpreter.setUp()
9          if connection between reactive gates then
10             eventManager ← new EventManager()
11             eventManager.setUp(MUTPath, DSLName)
```

ponents are instantiated and configured, including the Execution Engine, the OCL Interpreter, and the Event Manager.

Continuing with the main loop in Algorithm 1, after activating the test configuration, the test case behavior should be executed (line 4). The execution semantics of a behavior depends on its type. For instance, to execute a Message behavior (line 5), according to its source gate, the `argument` is treated differently. When the source gate belongs to a Tester Component, the `argument` is a request for the model under test (line 10), and when it belongs to a SUT Component, the `argument` is the expected result to be asserted (line 13).

Sending Requests to the SUT (shown in Algorithm 3): Depending on which gate of the SUT component is used for sending a request, the TDL Interpreter selects which external component (configured in Algorithm 2) should be used. Then, it checks whether the request can be accepted by the gate. Three cases are possible:

1. if the gate is a generic gate and the request is a model execution command (line 2), the configured engine should be used to run the command (line 3).
2. if the gate is an OCL gate and the request is an OCL query (line 4), the configured OCLInterpreter should be used to evaluate the query on the model (line 7). It should be noted that the query is evaluated on the model in its latest runtime state (line 6).
3. if the gate is a reactive gate and the request is an accepted event, the configured eventManager should be used to process the event.

Algorithm 3: Sending a request to the SUT

```
   Input:
   gate: the gate for sending request to SUT,
   request: the request to be sent
1  begin
2      if gate is generic gate & request is
         modelExecutionCommand then
3          engine.runCommand(request)
4      if gate is OCL gate & request is OCL query then
5          query ← createQuery(request)
6          MUTResource ← getMUTResource()
7          OCLInterpreter.runQuery(MUTResource, query)
8      if gate is reactive gate & request is accepted event then
9          event ← createEvent(request)
10         eventManager.processAcceptedEvent(event)
```

Asserting the Expected Output (shown in Algorithm 4): The TDL Interpreter asserts whether an expected output data is equal to the real output data (i. e., the data received from the model under test). Depending on which gate of the SUT component is used for the assertion, the output data has different semantics:

- *generic gate*: the expected output is indeed a specific runtime state of the model under test. So the TDL Interpreter retrieves the current state of the model from the context of the engine (line 3), and then checks whether the model state is as expected (lines 4-7).
- *OCL gate*: the expected output is the expected query evaluation result, so it should be checked against the result generated by the OCLInterpreter (lines 9-12).
- *reactive gate*: the expected output is an exposed event expected to be received from the model under test. Accordingly, the EventManager is requested to retrieve that event from the events exposed by the model (lines 14-15). If it retrieves nothing, the assertion fails (line 19).

In addition to the above-mentioned conditions, there are some specific cases that may lead to the interruption of the

Algorithm 4: Asserting expected output

```
   Input:
   gate: the gate for receiving data from SUT,
   expectedOutput: the expected output data to be asserted
   Output :
   verdict: the assertion result
1  begin
2      if gate is generic gate then
3          currentState ← engine.context.resource
4          if currentState == expectedOutput then
5              verdict ← PASS
6          else
7              verdict ← FAIL
8      if gate is OCL gate then
9          queryResult ← OCLInterpreter.result if
             queryResult.equals(expectedOutput) then
10             verdict ← PASS
11         else
12             verdict ← FAIL
13     if gate is reactive gate & expectedOutput is exposed
         event then
14         expectedEvent ← createEvent(expectedOutput)
15         exposedEvent ←
             eventManager.getExposedEvent(expectedEvent)
16         if exposedEvent != NULL then
17             verdict ← PASS
18         else
19             verdict ← FAIL
```

test case execution. This happens for instance when the test system sends a syntactically wrong OCL query to the SUT, or the exchanged event does not conform to the behavioral interface of the xDSL specified by the test configuration, or when the running external component throws some exception. In these cases, the TDL Interpreter interrupts the test case execution and sets the verdict to INCONCLUSIVE.

6 Interactive Debugging and Mutation Analysis for TDL Test Cases

After defining and running a test suite for a given model, two important concerns remain for the domain expert: being able to *localize* the defects of failed test cases, and being able to *measure* how well the test suite is in finding faults. In this section, we address these needs with facilities for interactive debugging and mutation analysis, respectively.

6.1 Interactive Debugging

A failed test case is essentially an *alert* for the domain expert which tells there is a defect in the model causing the failure. For trivial test cases, the test report may provide adequate information about the cause of failure. However, fault localization can be more difficult for more complex test cases such as event-driven ones which can involve complex sequences of events exchanged with the model. In such cases, *interactive debugging* is a technique commonly used in the realm of software testing, allowing to execute and observe the SUT behavior one step at a time. However, it has not yet been leveraged for model testing.

In what follows, we first define what is interactive debugging. Then, we explain what are the obstacles preventing the use of interactive debugging with TDL test cases. Finally, we present how we overcome these obstacles, and thus provide interactive debugging for TDL test cases.

6.1.1 Definition of Interactive Debugging

Interactive debugging involves manual control and observation of an execution with the help of an interactive debugger. Such debugger provides services to *pause* and *unpause* the execution through *breakpoints*—i. e., conditions upon which the execution must be paused, such as "reaching a specific model element"— and prepares information to observe the execution, such as the current stack of method calls or the values of all existing variables. An execution can be represented as a sequence of execution steps (e. g., a sequence of statements), and a step may itself contain a sequence of inner steps (e. g., method calls, leading to more statements). Based on this representation, an interactive debugger also provides a common set of operators to perform step-by-step observation of an execution, such as:

- The *resume* operator, to continue the execution until a breakpoint is reached.
- The *step over* operator, to continue the execution until the end of the current step or until a breakpoint is reached, hence ignoring the possible inner steps.
- The *step into* operator, to continue the execution until either some inner step is reached (if any) or when the current step ends.

Note that a typical interactive debugger offers other services as well, such as conditional breakpoints or the ability to query/change the model runtime state. Yet, this paper focuses only on the above-described stepping operators— which are the most essential services of an interactive debugger— and leaves other debugging services for future work.

6.1.2 Requirements for Debugging TDL Test Cases

In the context of software testing, most of the popular testing frameworks (e. g., JUnit) are compatible with interactive debugging facilities (e. g. `jdb`). Among other possibilities, this allows the tester to perform a step-by-step observation of the SUT behavior as triggered by the test case. But for

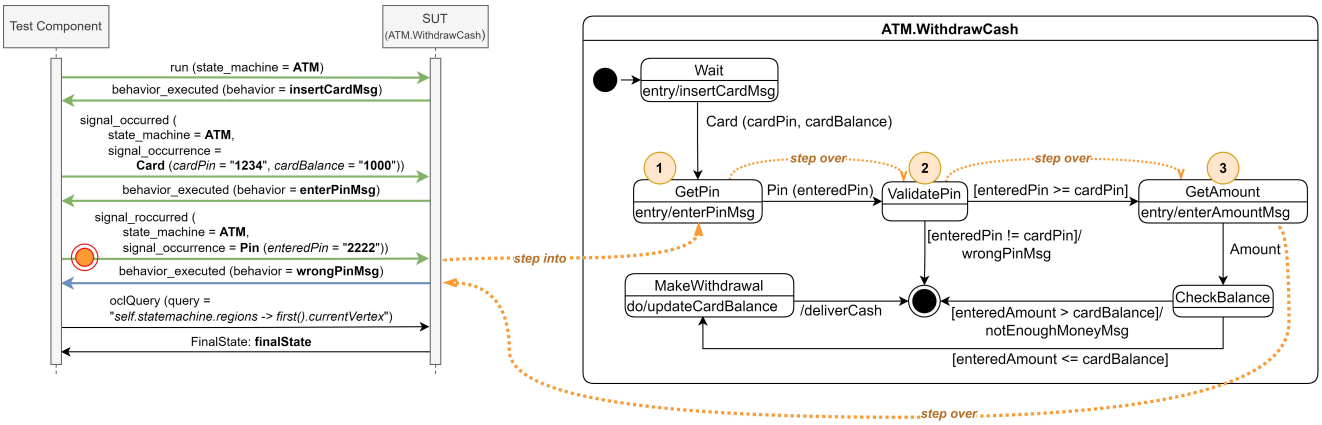


Fig. 8 A sample scenario of performing interactive debugging for the running example

this to work, it must be possible to execute step-by-step not only the SUT, but also the test case itself. In other words, it must be possible to perform interactive debugging for both the test case and SUT *in unison*. When the test case and its SUT are both implemented using the same language (such as Java programs and their JUnit tests), this is trivial to achieve using a single debugger instance, since both the SUT and the test case are then executed as one single executable program.

However, in the context of this paper, the test case and the SUT are two different executable models conforming to two different languages. This means we need to (1) be able to debug the executable model itself, and then (2) to initialize two debugger instances at the same time, one for the test case and another for the model under test, while making sure the debugging services remain consistent when used in two different debuggers, and coordinating the communication between the two debuggers. To the authors' knowledge, the first matter is already addressed for both EMF [4, 5] and UML [6] models, and we use the interactive debugging approach of [5][4] as it supports the xDSLs considered in the context of this paper. More specifically, their approach can be configured for a specific xDSL and then can be used to debug its conforming executable models. However, the second challenge is still open and the remainder of this subsection explains our proposal to resolve it.

6.1.3 Adapting Interactive Debugging for TDL

When running a TDL test case with an interactive debugger, as soon as the execution reaches a point where the test case makes a request to the model under test (e. g., a TDL message sending an event to the reactive model), one can expect to be able to "jump" from the debugger of the TDL test case to the debugger of the model under test, and to switch to observing the model's behavior. More precisely, this can be expected when the modeler either sets a breakpoint inside the model under test or wishes to *step into* the processing of the request sent to the model by the test case (e. g., an event). To meet these expectations in our approach, we make the following minor adaptations to common interactive debugging services for debugging TDL test cases:

- The *resume* operator: It continues the test case execution until a breakpoint is reached either in the test case or in the model under test.
- The *step over* operator: It continues the execution until the end of the current step or until a breakpoint is reached in the test case or in the model under test.
- The *step into* operator: It continues the execution until either some inner step is reached (if any) or the current step is ending. If in the current step, the test case sends a request to the model under test, the *step into* operator pauses the execution inside the model under test at the very beginning of processing the sent request.

For example, Figure 8 illustrates an interactive debugging scenario for the running example using our redefined debugging services. Here we see a situation where the modeler has set a breakpoint (shown as a filled colored circle) in the faulty TDL test case (previously shown in Figure 6), on the TDL message that sends a signal_occurred event for the Pin signal to the *ATM.WithdrawCash* state machine. When the test case execution reaches this TDL message, it pauses because of the breakpoint. The modeler may wish to investigate how this event will be processed in the state machine. So by using the *step into* operator, the execution pauses at the beginning of processing said event by the ATM state machine i. e., the GetPin state (label 1).

Afterward, by using the *step over* operator in the model debugger, the transition to the ValidatePin state fires because the state machine has received a signal_occurred event for the Pin Signal from the test case which is the Trigger of this transition (label 2).

Using the *step over* once more allows the modeler to observe that instead of the transition to the FinalState, the tran-

[4] One of our authors is involved in [5]

sition to the GetAmount state traverses, hence discovering the defect in the constraint of this transition. Accordingly, the execution enters the GetAmount state (label 3) and the processing of the event ends because there is no more transition to traverse.

Finally, using the *step over* operator, the test case debugger resumes, so the next TDL message can be executed (i. e., the TDL message after the breakpoint).

6.1.4 Initialization and Coordination of Two Interactive Debuggers

As previously mentioned, debugging a TDL test case requires two interactive debugger instances, one for the test case and one for its model under test. This subsection explains how we spawn and coordinate them using a sample scenario shown in Figure 9. The domain expert starts the process by requesting to debug a TDL test suite containing at least one TDL test case. This results in initializing a debugger for the test suite, preparing the TDL Interpreter, and pausing the execution where reaching the first breakpoint (if any). In the scenario of Figure 9, it pauses at the very beginning of the test suite execution as we configured a breakpoint there. Then, the domain expert can use the *step over* service of the debugger to start the execution of the first test case. As described in Section 5, for test case execution, the TDL Interpreter first activates the test configuration of the test case. For example, when the test case is event-driven and so its tested model is reactive, the TDL Interpreter configures an instance of an Event Manager. Hereupon, the internal behavior of the test case can be executed step-by-step using the services of the test case debugger, such as *step into*.

When the test component requests an execution in the model under test, if the domain expert wants to observe the model's behavior upon receiving that request, a second debugger is required to be initialized for the model. To do this, we added new functionalities to our *TDL Interpreter* component. As shown in Figure 9, at the first time that the domain expert chooses the *step into* operator (in the test case debugger) when the test component sends a request to the

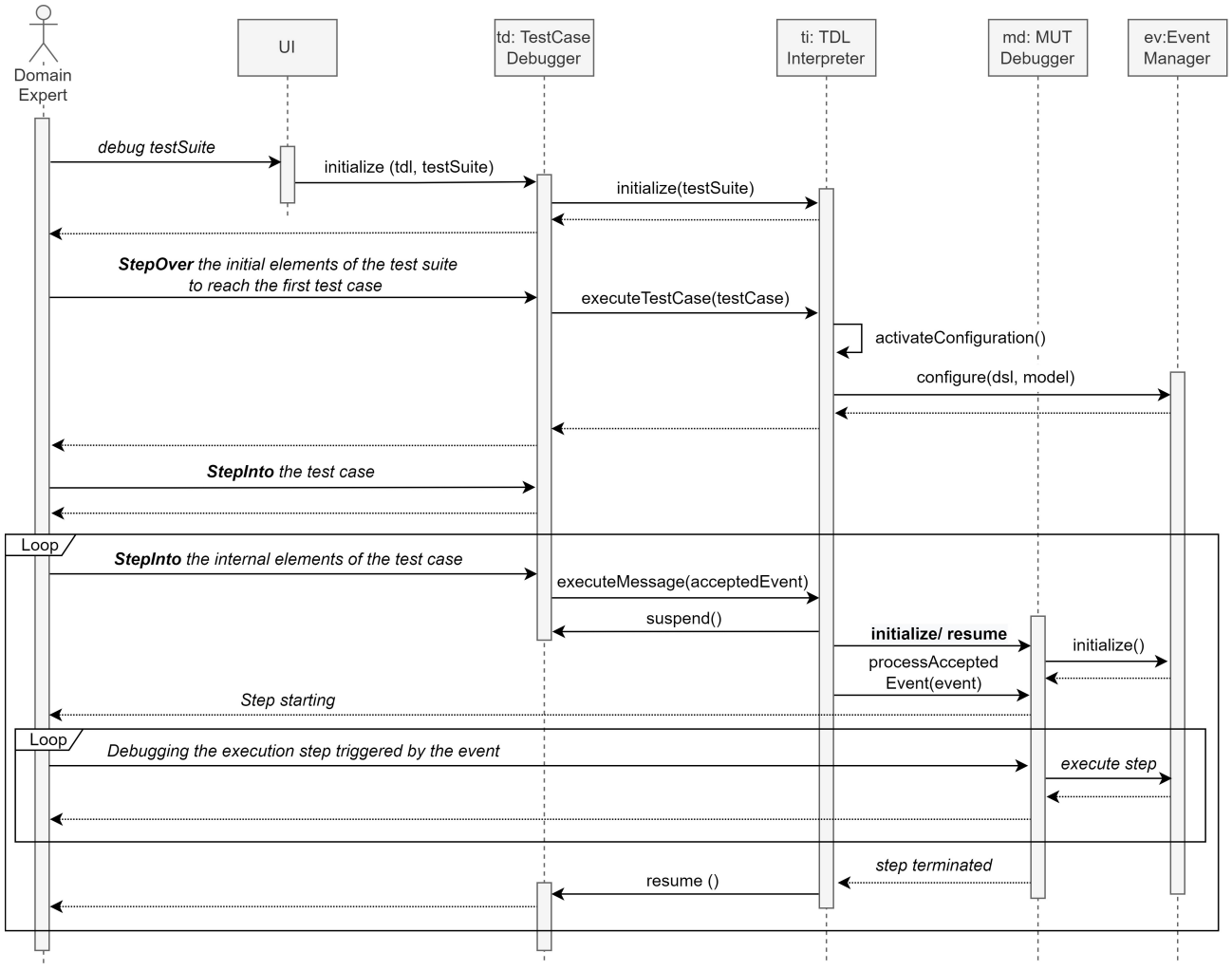


Fig. 9 One possible interactive debugging scenario for an event-driven TDL test case written for a reactive model

model (e. g., an accepted event), the TDL Interpreter initializes a debugger for the model. Hereafter, the TDL Interpreter pauses and resumes the model debugger according to the debugging services chosen by the domain expert in the test case debugger, based on their redefined semantics presented in Section 6.1.3. It also deactivates the test case debugger for the active time of the model debugger to ensure their consistency.

6.2 Test Quality Measurement

Measuring the quality of a test suite is often used to decide whether the test suite should be improved, and how much effort should be put into this endeavor. A popular measurement techniques is *mutation analysis* which follows this idea: if we inject artificial faults into the SUT, an existing test suite that can find those faults is probably good enough at discovering real faults [8]. The artificial faults are defined in the form of mutation operators which can perform small syntactic changes and are systematically applied on the SUT to produce a set of mutants (i. e., faulty programs). Afterward, the test suite is run on each mutant. If there is at least one test case in the test suite that its execution result is different for the SUT and the mutant, we conclude the test suite has detected the fault of the mutant, and the mutant is said to have been 'killed' by the test suite. Finally, a mutation score is calculated that is the percentage of killed mutants among all generated mutants. This mutation score is a criterion for measuring the quality of the test suite.

To provide mutation analysis in a generic model testing approach, four features are required: (1) a definition of mutation operators for the considered xDSL; (2) a process to generate mutants out of models conforming to the considered xDSL; (3) a way to execute the considered test suite on each mutant; and (4) a way to calculate the mutation score for the test suite. Recently, a framework named WODEL-Test was proposed by Gómez-Abajo et al. and is able to support most of the above features [15]. More specifically, WODEL-Test allows a language engineer to define mutation operators for her/his xDSL if the abstract syntax is provided as a metamodel. Then, it automatically generates mutants for the models conforming to that xDSL by applying the defined mutation operators.

However, WODEL-Test does not provide any testing facility, and thus fail at providing feature (3). It indeed assumes there is an existing testing framework for the given xDSL which allows writing test suites for the conforming models and provides an interface to run such test suites and get the result. Based on this assumption, WODEL-Test generates an environment for the domain experts to run their written test suites on the generated mutants and to get their mutation scores. As our proposed testing approach realizes this assumption, we can offer a complete mutation testing framework for any xDSL through an integration with WODEL-Test. Figure 10 shows how we achieved this integration, where the same roles and artifacts as our approach (Figure 5) are involved with only one additional task for the language engineers to define mutation operators for their xDSLs. As a result, the domain experts can write TDL test suites for the conforming models and evaluate their quality.

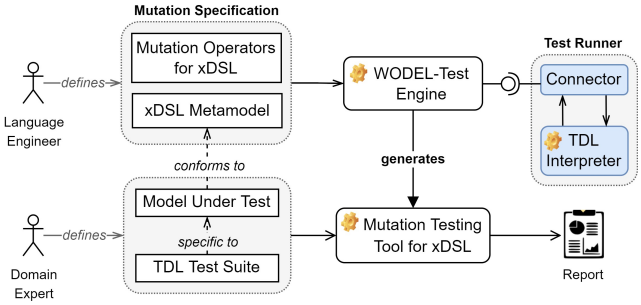


Fig. 10 An overview of the integration of the TDL Interpreter with WODEL-Test [15]

As mentioned above and shown in Figure 10, WODEL-Test requires a test runner implementing a specific interface that determines how to run a test suite on a mutant and how to decide whether the given mutant is killed by the test suite. Accordingly, we connected our TDL Interpreter to the WODEL-Test engine by defining a connector which implements the interface and performs three operations: (1) it receives a TDL test suite and a mutant from the WODEL-Test and runs the test suite on the mutant using the TDL Interpreter; (2) it receives the test execution result from the TDL Interpreter and sets the mutant as 'killed' if there is at least one test case in the test suite that is failed on the given mutant; and (3) it provides the final mutation testing results in conformance to the WODEL-Test result templates.

It is worth mentioning that for this integration, we added some extra features in the proposed TDL Interpreter. As described in Section 4, the TDL Interpreter runs a TDL test case on the model that is persisted in the path specified in the test configuration of the test case. Consequently, to execute the test case on another model, we need to modify the test configuration. However, for mutation testing, a TDL test case must be run on several models i. e., the original model and the mutants generated for it, without modifying the test case definition—including the test configuration. To this end, we provide an optional service in the TDL Interpreter to be able to run a test case on a specific model while ignoring the model path specified in the test configuration.

7 Tool Support

We implemented each component of our proposed approach as part of the GEMOC Studio [4], a language and modeling

workbench defined on top of the Eclipse Modeling Framework (EMF). A base implementation of the TDL Library Generator was reused from our previous work [20], and was extended with the new contributions of the present paper—mainly, a model transformation able to translate behavioral interfaces into TDL. We used the implementation of Leroy et al. [24] for the behavioral interface definition (i. e., also a part of the GEMOC Studio) and we implemented the transformation in Java. A base implementation of the TDL Interpreter, written in Xtend [9], was also reused from our previous work [20]. We improved its operational semantics and we integrated it with an existing Event Manager of the GEMOC Studio [24].

Next, the Interactive Debugging component uses the model debugging framework of the GEMOC Studio [4, 5] for the initialization of two debugger instances, and the Eclipse debug platform for managing their communication and synchronization, all implemented in Java. For the Test Quality Measurement component, as the WODEL-Test framework is also implemented using EMF technologies [15], we easily integrated it into our testing framework. More specifically, we implemented a connector in Java (shown in 10) which connects the WODEL-Test engine to our TDL Interpreter.

Figure 11 displays three screenshots of the provided facilities running in the GEMOC studio modeling workbench for the *ATM.WithdrawCash* state machine. The source code is available on a public GitLab instance[5]. A screenshot of the testing tool is shown in Figure 11a. Using the provided icons in the toolbar and the menubar, we executed the TDL Library Generator for the xPSSM DSL and it successfully generated an xPSSM-specific TDL library (label 1). In the middle, the event-driven TDL test case of Listing 4 is shown (label 2) which is failed on the ATM state machine (label 3) because the model has exposed a behavior_executed event for the enterAmountMsg behavior (label 4) while the expected output is wrongPinMsg behavior.

Figure 11b shows the usage of the interactive debugging facility for the running example. It displays two debugger instances, one for the test case (label 1) and another for the ATM state machine (label 2), both running using GEMOC execution engines (label 3). Running the test case in debug mode, we chose the *step into* operator of the test case debugger where the test case wanted to send a signal_occurred event for the Pin signal to the ATM state machine. This paused the test case debugger on the first of the next TDL Message (i. e., asserting a behavior_executed event for the WrongPinMsg behavior) and enabled a debugger for the model under test. Using the stepping operators of the ATM debugger (label 2), we observed when the ATM state machine has received the said event from the test case, the transition from the ValidatePin to the GetAmount state has been fired and the enterAmountMsg behavior has been executed. The GEMOC debugging tool also provides the values of all existing variables for each debugger instance. For example, we selected the ATM debugger, and at the bottom left, we can see the values of its variables (i. e., the last executed vertex and the execution status of all states).

Figure 11c shows how mutation analysis appears in the tool. Here, we analyze a TDL test suite (containing four test cases) tailored for the correct version of the ATM state machine and the result is shown in Figure 11c. Note that the shown mutation operators for the xPSSM DSL are explained in the next section. Under the state machine project (label 1), one folder per mutation operator exists, each containing mutants generated by WODEL-Test by applying that operator. We can see that the TDL test suite was exectued (label 2) on all models i. e., the original model and all the generated mutants. The global result (label 3) reports that 186 mutants are generated by applying 90 % of mutation operators (18 out of 20) and the mutation score for the considered TDL test suite is 67.2 %. The tool also provides information about the test suite execution result for each mutant (label 4).

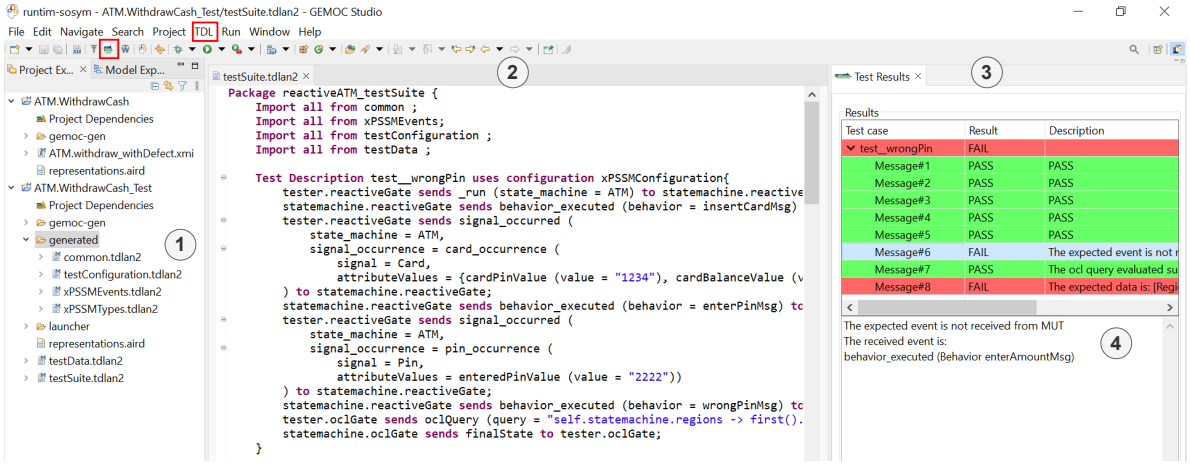
[5] https://gitlab.univ-nantes.fr/naomod/faezeh-public/xtdl/-/tree/SoSym2021

8 Evaluation

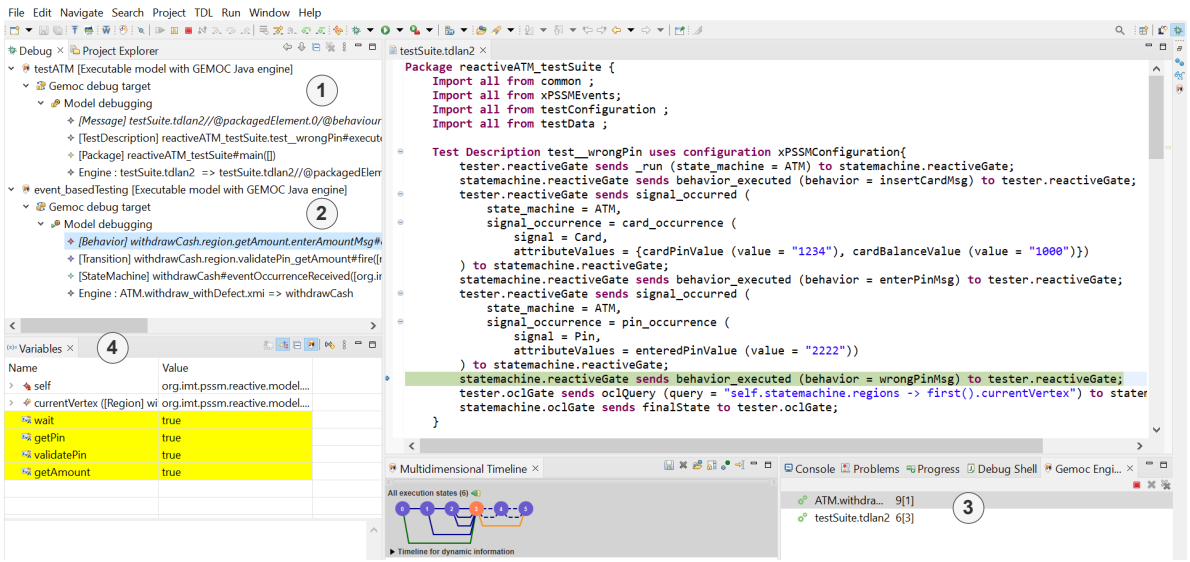
The main goal of our approach is to be *generic* in terms of providing testing and debugging facilities for any reactive xDSL that follows the definitions given in Section 2.1. To evaluate the *genericity* of our approach and to investigate whether it fulfills each of the requirements listed in Section 2.4, we designed and performed an empirical evaluation which is presented in this section.

8.1 Experiment Setup

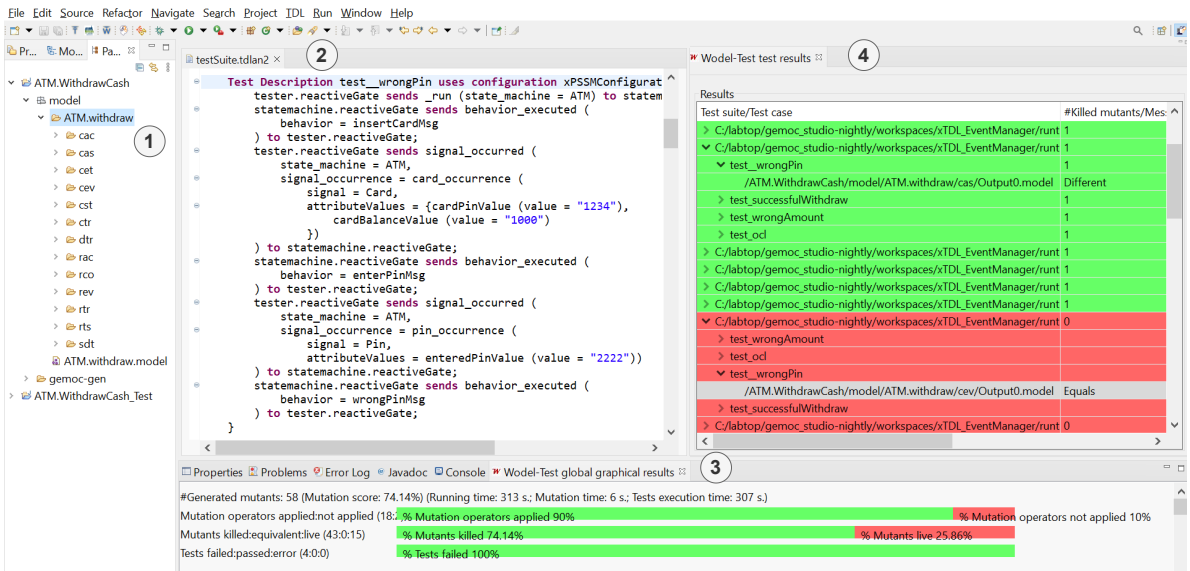
Considered xDSLs In our experiment, we used the prototype presented in Section 7 for two reactive xDSLs. The first one is the xPSSM DSL already presented in Section 2.1.1, and the second one is called *xArduino*. It is a specific DSL for modeling Arduino boards and their behavior. An Arduino model contains a Board representing a physical board, and a Sketch that is executed on the board. A Board comprises various modules such as LEDs, sensors, and buttons. A Sketch is a block of instructions, each performing a specific behavior such as turning on an LED or suspending the execution for a specified duration. Using the behavioral interface of the xArduino DSL, an external tool can communicate with a running xArduino model. For example, if we define an xArduino model comprising a button and an LED which blinks when the button is pressed, an external tool can request for pressing the button during execution and will be notified each time the LED turns on/off.



(a) TDL testing facilities



(b) Interactive debugging facilities



(c) Mutation analysis (using WODEL-Test facilities [15])

Fig. 11 Screenshots of the provided tools running in the GEMOC studio modeling workbench for the *ATM.WithrawCash* state machine

Table 2 The evaluation data

		xPSSM	xArduino
xDSL Size	Abstract syntax size (n. of EClasses)	35	59
	Semantics size (LoC)	975	768
	Behavioral Interface size (n. of Events)	4	6
Tested Models	Number of tested Models	65	5
	Size range of tested models (n. of EObjects)	13 - 154	15 - 41
Test Artifacts	TDL Library size (LoC generated)	178	252
	Total n. of test cases	217	30
	Size range of test suites(LoC)	25 - 1311	30 - 132

Table 2 shows the size of each xDSL as the number of EClasses of its abstract syntax, the number of lines of code (LoC) of its operational semantics, and the number of events of its behavioral interface. According to their values for each xDSL, xArduino has a larger abstract syntax and behavioral interface, but a smaller operational semantics. The source code of both xDSLs is available on a public GitLab server.

Considered Models For each xDSL, we need a set of conforming models to be tested. For the xPSSM DSL, the PSSM standard provides a set of UML state machines, each with a small test suite for asserting that a PSSM implementation executes the models as expected, indeed in compliance with the standard [36]. We used a subset of them (60 models) which represent an event-driven behavior using solely state machines. In addition, the *ATM.WithdrawCash* state machine presented in Section 2.1 is considered in the experiment, and we manually defined four larger state machines[6] for a total of 65 xPSSM models (60+5). As written in Table 2, the size of the considered xPSSM models ranges from 13 to 154 EObjects. For the xArduino, we manually defined five models ranging from 15 to 41 EObjects in size.

8.2 Evaluating Genericity

To evaluate the genericity of the approach, we applied it on both considered xDSLs following the same process. First, we executed the TDL Library Generator component for each xDSL and it successfully generated a domain-specific library for each of them. The number of lines of code for the xPSSM-specific TDL library is 178 and for the xArduino is 252 (also shown in Table 2). As the TDL library is generated from the xDSL definition excluding the execution rules, the size of the generated library for the xArduino is larger than that of for the xPSSM DSL.

Second, using each generated TDL library, we wrote a set of TDL test cases for each considered model. In total, 217 TDL test cases for the xPSSM models (60 of them are transformed from the standard PSSM test suites [36] to TDL) and 30 test cases for the xArduino models. The number of lines of code for the test suites is shown as a range in Table 2. The smallest and the largest test suites are written for the xPSSM models, with 25 and 1311 LoC, respectively. All the tested models and their test cases are publicly accessible on a public GitLab server.

Lastly, we executed the TDL test cases on the models using the TDL Interpreter component. For all the test cases, the test verdicts were set and the test results were reported using the graphical view provided by our tool. We also manually verified that we obtain the expected verdict for each test case. In conclusion, we successfully used the proposed approach for two reactive xDSLs whose abstract syntaxes represent *different domains* and whose execution semantics implement *different behavioral interfaces*. Therefore, we can conclude that our approach is not tied to only one specific xDSL, and thus provides a certain degree of genericity.

8.3 Evaluating Requirement Fulfillment

One of the objectives of our empirical evaluation is to investigate the ability of the approach in fulfilling the requirements listed in Section 2.4.

Fulfilling Req.1 In the first requirement, the testing approach is requested to support the events specified in the behavioral interface of an xDSL as test data types when writing test cases for its conforming models. As explained above, the TDL Library Generator successfully generated a domain-specific TDL library for each considered xDSL. This library contains an *xDSL-Specific Events Package* which provides the definition of the xDSL's behavioral interface in TDL. We also explained that we used the generated TDL library of each xDSL to write event-driven TDL test cases for their conforming models. Therefore, our proposed testing approach fulfills its first requirement.

Fulfilling Req.2 The second requirement concerns the execution of event-driven TDL test cases on the reactive models. We described in Section 5 that our testing approach provides an integration between the TDL Interpreter and an Event Manager to realize this requirement and we explained in the previous subsection that we successfully executed the test cases of both xPSSM and xArduino reactive models. Accordingly, we conclude that our proposed approach fulfills the second requirement as well.

Fulfilling Req.3 The third requirement is related to providing facilities for test case failure diagnosis. For realizing this requirement, our testing approach proposes inter-

[6] We used the samples from: https://www.uml-diagrams.org/state-machine-diagrams.html

active debugging facilities for TDL test cases. We used interactive debugging on both a set of failed TDL test cases of non-reactive models (four test cases from our previous work [20]) and a set of failed event-driven TDL test cases of reactive models (such as the running example shown in Figure 6). In both cases, we successfully debugged the model under test in unison with its test case, which ultimately helped us to localize the defect of the model. This shows the approach fulfills the third requirement.

Fulfilling Req.4 In the last specified requirement, the adoption of mutation analysis for measuring the quality of the TDL test suites is needed. Our testing approach offers such facility by integration with the WODEL-Test mutation testing framework [15]. To evaluate this integration and demonstrate its usage, we performed mutation analysis of the considered xPSSM test suites as described below.

At first, we defined a set of mutation operators for xPSSM, listed in Table 3. A mutation operator can be applied on an xPSSM model, if the operator's requirement is realized by the model (the third column in Table 3). The definition of these mutation operators is driven from existing work referenced in the fourth column of the table and we implemented them using the WODEL DSL [14]. Afterward, we ran mutation analysis on the TDL test suites that we have written in Section 8.2 for 65 xPSSM models. This run includes three steps for each xPSSM model: (1) generating mutants for the xPSSM model by applying the mutation operators; (2) executing the TDL test suites provided for the model on both the seed model and its generated mutants; and (3) reporting the mutation analysis result, such as percentage of the applied mutation operators, number of the generated mutants, mutation score (i. e., percentage of killed mutants), and test execution result for each model. The first step is completely performed by the WODEL tools, and the rest is done by our integration of the WODEL and TDL tools.

The three charts of Figure 12 demonstrate our analysis results. As shown in Figure 12a, from 20 mutation operators described in Table 3, 13 were applicable on 3 of xPSSM models, 17 on 17 models, 18 on 19 models, and 19 on the rest of 26 models. The number of generated mutants for each xPSSM model is provided in Figure 12b which ranges from 28 to 664, a total of 12674 mutants. Finally, the mutation score for the 65 TDL test suites of xPSSM models is presented in Figure 12c which ranges from 43.54 % to 74.78 %. Therefore, we claim that the approach succeeds in fulfilling the fourth requirement.

8.4 Threats to Validity

The fact that we only used the approach for two xDSLs threatens the validity of its genericity. We tried to use two xDSLs that are very different from each other regarding their supporting domain and their execution semantics. However, there is a need to explore the approach for more reactive xDSLs made for different and more complex domains.

Our proposed approach aims to support the domain experts in writing and executing test cases for their executable models. To validate its usability for the domain expert, a user study should be performed. Accordingly, a threat exists regarding the approach usability and we consider it for our future work. However, as our approach uses TDL which is a standard testing language particularly defined for the non-technical testers, and as we support using the domain concepts in writing TDL test cases, we tried to take the usability feature into account.

9 Related Work

Existing related work can be categorized either as ad-hoc testing approaches made for specific DSLs, or generic testing approaches applicable to many DSLs.

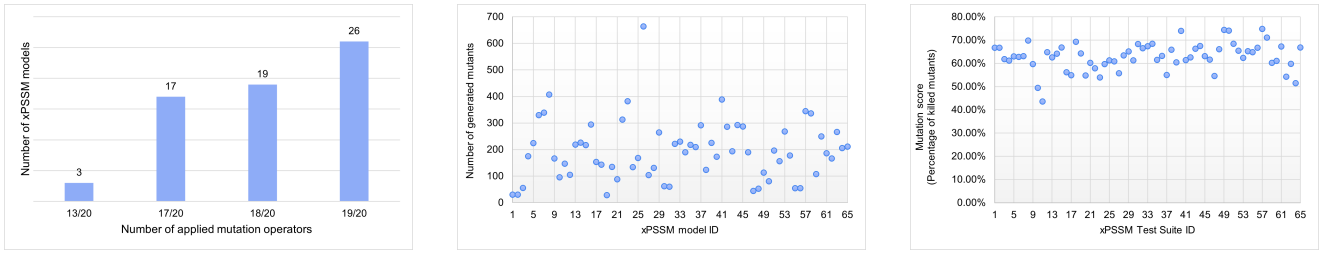
9.1 Testing Approaches for Specific DSLs

A specific methodology for designing and early testing of executable statecharts is proposed in [28] which supports four popular testing techniques: Test-Driven Development (TDD), Behavior-Driven Development (BDD), Design by Contract (DbC), and property statecharts for monitoring the violation of behavioral properties during model execution. In the design phase, several tasks have to be performed, some of which are required for the testing phase such as defining the execution scenarios using the BDD approach, implementing the mapping between the steps of the scenarios and the statechart test primitives (in Python), and writing unit tests (in Python). The scenarios and the unit tests will then be executed on the statecharts to validate their behavior. Although they support a complete process of designing and validating statecharts, the testing activities should be performed by a technical tester as coding in Python is required.

Hili et al. propose an approach for interactive monitoring of real-time and embedded systems modeled using UML Real-Time (UML-RT) [17]. Their approach enables different external components such as tools for data collection, animation, simulation, analysis, adaptation, and control to monitor the execution of the code generated from a UML-RT model. As one of their case studies, they show how the approach can be used to perform functional steering. Therefore, although they do not provide any testing approach for writing test cases, they show how the approach can be applied for testing purposes. Moreover, they are focusing on monitoring the behavior of the generated code while we are focusing on testing the executable models to ensure the correctness of their behavior as early as possible.

Table 3 The mutation operators for the xPSSM DSL

Symbol	Description	Requirement from the model	Reference
ccs	Creates a connected state	Containing at least one state	[37, 25, 15]
ccfs	Creates a connected final state	Containing at least one state	[25, 15]
ctr	Creates a transition with random source and target states	Containing at least two states	[25, 15]
ctr2	Creates a transition with source but without target state	Containing at least one state	[12]
rtr	Removes a transition	Containing at least one transition	[37, 12, 25, 39, 15]
dtr	Duplicates a transition	Containing at least one transition	[39]
rst	Removes a state and adjacent transitions	Containing at least one state	[12, 15]
cis	Changes the initial state to a different one	Containing at least two initial states	[37, 12, 25, 15]
cfs	Changes the final state to a different one	Containing at least two final states	[25]
cst	Changes the source state of a transition	Containing at least one transition and two states	[39]
rts	Changes the target state of a transition	Containing at least one transition and two states	[12, 39, 15]
sdt	Swaps the direction of a transition	Containing at least one transition	[25, 15]
rev	Removes a trigger	Containing at least one trigger	[37, 12, 25]
cev	Creates a trigger and set it to a transition	Containing at least one transition	[37, 12, 25]
cet	Changes the trigger of a transition	Containing at least one trigger	[37, 12, 25]
rac	Removes a behavior	Containing at least one behavior	[37, 12, 25]
cac	Creates a behavior and set it to a transition	Containing at least one transition	[37, 12, 25]
cat	Changes the behavior of a transition	Containing at least one transition with a behavior	[37, 12, 25]
cas	Creates a behavior and set it to a state	Containing at one state	[25]
rco	Removes a constraint	Containing at least one transition with a constraint	[39]



(a) Number of applied mutation operators for all xPSSM models

(b) Number of generated mutants for each xPSSM models

(c) Mutation score for each xPSSM test suite

Fig. 12 The result of mutation analysis of xPSSM models

To tackle the inherent complexity of testing domain intensive cloud applications, a configurable test DSL is proposed in [38]. Given an abstract definition for a cloud application (indeed for its User interface, user interactions, data setup, environment, and platform configuration) using the domain concepts, it generates a specific test DSL and a testing toolset named Legend for authoring, executing, and debugging test cases for cloud applications [21].

In the context of measurement systems, a specific DSL named Sequencer is used in the NASA awarded measurement system (DEWESoft) which enables adjusting measurements and creating measurement procedures. To provide testing support for the Sequencer DSL, a specific testing framework namely Sequencer Testing Tool (SeTT) is proposed in [22]. The SeTT tool enables the domain expert to define test cases for each part of the measurement system. It indeed allows augmenting test elements such as assertions into the Sequencer models.

To define test cases for the executable business processes that are modeled using Web Services Business Process Execution Language (WS-BPEL) or BPMN2, a specific testing approach is proposed in [26]. They use a metamodel extension technique to add test-specific elements (e. g., assertions) to the BPMN metamodel. To ensure the test models have deterministic behaviors, they enforce some control-flow restrictions. The domain expert can define test cases as BPMN models in which there is one *Pool* describing the process under test and other *Pools* specifying the test case behavior. The *Pools* communicate with each other by exchanging messages To execute such test models, the technical information for running the physical operations of the process under test must be provided in advance.

A sizable amount of works are proposing testing approaches for the fUML [33] which are described below. A BDD framework enables describing the requirements as executable user stories and the acceptance criteria as executable scenarios attached to the user stories. In [23], a BDD framework is proposed for the fUML by defining a UML profile for BDD and a BDD library comprising executable commands required when describing fUML scenarios. The

framework allows the domain expert to define fUML models following a BDD approach, meaning that she/he first defines executable fUML stories and scenarios and then describes the fUML models satisfying them.

In [2], a testing approach is proposed for fUML where the behavioral scenarios of a system are first described using UML sequence diagrams enriched with timing properties that are described in UML MARTE constraint language. These diagrams describe the communications between the different components of a system, and each component is itself described using fUML activity diagram. In this work, a testing tool is provided which automatically evaluates the conformance of the fUML activities to the sequence diagrams and their timing constraints. In addition, they generate test input data from the sequence diagrams and use them to test the behavior of the activity diagrams automatically.

In [30], a functional testing framework is proposed to validate the behavior of fUML models. For describing test cases, they provide an executable test specification language that supports using temporal expressions for the precise selection of the runtime states to be asserted, using OCL queries for specifying complex assertions on the runtime states of a system that behaves concurrently, and verifying the execution order of the activity nodes for concurrent systems.

In [18], a simulation and test generation approach is proposed for the fUML activity diagrams containing Alf[7] code [18]. At first, the fUML models are translated into Java code. Afterward, the test input data are generated automatically from the Java code, enabling an exhaustive simulation of the fUML models. Finally, using the provided simulation, the test cases along with the test oracle are auto-generated satisfying 100 % coverage of the Java code.

To sum up, DSL-specific approaches promote usability, as they enable the domain experts to describe test cases using the system description language that is familiar to them. Nevertheless, they lack reusability since a new test language must be engineered for each new DSL. In contrast, our approach provides generic testing solutions reusable for a wide range of xDSLs.

9.2 Generic Testing Approaches

When a grammar-based DSL has a translational semantics, if the target language (i. e., a general-purpose language) provides a unit testing framework (e. g., JUnit for Java), then the work of Wu et al. provides a unit testing framework for that DSL [41]. It requires the language engineers to define the mapping algorithms between the testing actions of their DSL and the target GPL. Accordingly, the framework can translate test cases from DSL code to GPL which enables using the GPL testing tools for executing test cases on the generated GPL code of the model under test. It also translates the test results from the GPL code to the DSL, to report the result using the domain concepts. Therefore, this approach is useful for compiled DSLs and performs testing at the code level, while we provide testing facilities for the interpreted DSLs and the test cases are run at the model level.

Meyers et al. propose a generic testing approach for xDSLs with discrete-event semantics i. e., reactive xDSLs [29]. Given an input xDSL, it generates an xDSL-Specific testing language by extending the abstract syntax of the xDSL with a limited set of testing features. To execute each test case written using this language, the operational semantics of the xDSL must be instrumented specifically for it. Instrumentation means new execution rules (i. e., for test case execution) must be added to the xDSL's execution rules. This in turn requires (1) using the same approach for implementing both execution rules, and (2) the language engineer should enrich the abstract syntax of the xDSL with event-related concepts to specify where new rules must be added. In contrast, our approach does not require changing the xDSL definition, hence being easily applicable to any reactive xDSL. In addition, we use a standard testing language (i. e., TDL) for writing test cases and we offer two analysis techniques to help the domain expert in performing testing activities.

10 Conclusion and Future Work

A reactive xDSL with testing support enables its users to validate the behavior of their reactive models as early as possible. In this paper, we proposed a *generic* testing approach for reactive xDSLs with discrete-event operational semantics using the TDL standard testing language. Given a reactive xDSL, our approach offers services for the definition, execution, debugging, and quality measurement (i. e., based on mutation analysis) of the test cases for the conforming reactive models. We evaluated the genericity of the approach by its application for two different xDSLs. In conclusion, we observed that our generic testing approach for xDSLs advances the tool support for existing as well as emerging xDSLs.

We have identified several interesting research directions for the future, such as performing a user study to evaluate the usability feature of the approach for the domain experts, and extending the approach to support integration testing of compositional models i. e., models conforming to several interconnected xDSLs. In addition, benefiting from our mutation analysis support, we can develop efficient test generation techniques for reactive xDSLs since mutation analysis allows us to evaluate if a test generator works efficiently [1].

Acknowledgements 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. We would

[7] Action language for fUML

like to appreciate the great help of Dr.Pablo Gómez-Abajo for the integration of our work with the WODEL-Test framework.

References

1. J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering*, 32(8):608–624, 2006.
2. M. Arnaud, B. Bannour, A. Cuccuru, C. Gaston, S. Gerard, and A. Lapitre. Timed symbolic testing framework for executable models using high-level scenarios. In *Complex Systems Design & Management*, pages 269–282. Springer, 2015.
3. R. Bendraou, B. Combemale, X. Crégut, and M.-P. Gervais. Definition of an eXecutable SPEM 2.0. In *14th Asia-Pacific Software Engineering Conference (APSEC)*, pages 390–397. IEEE Computer Society, 2007.
4. E. Bousse, T. Degueule, D. Vojtisek, T. Mayerhofer, J. Deantoni, and B. Combemale. Execution framework of the gemoc studio (tool demo). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, page 84–89. Association for Computing Machinery, 2016.
5. E. Bousse, D. Leroy, B. Combemale, M. Wimmer, and B. Baudry. Omniscient debugging for executable dsls. *Journal of Systems and Software*, 137:261–288, 2018.
6. F. Ciccozzi, I. Malavolta, and B. Selic. Execution of uml models: a systematic review of research and practice. *Software and Systems Modeling*, 18:2313–2360, 2019.
7. J. Deantoni. Modeling the Behavioral Semantics of Heterogeneous Languages and their Coordination. In *Architecture Centric Virtual Integration (ACVI)*. Julien Delange and Jerome Hugues and Peter Feiler, 2016.
8. R. DeMillo, R. Lipton, and F. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.
9. S. Efftinge, M. Eysholdt, J. Köhnlein, S. Zarnekow, R. von Massow, W. Hasselbring, and M. Hanus. Xbase: Implementing domain-specific languages for java. *SIGPLAN Notices*, 48(3):112–121, 2012.
10. ETSI ES 203 119-1. Methods for testing and specification (mts); the test description language (tdl); part 1: abstract syntax and associated semantics, 2020. URL `https://tdl.etsi.org/index.php/downloads`.
11. ETSI ES 203 119-6. Methods for testing and specification (mts); the test description language (tdl); part 6: Mapping to ttcn-3, 2020. URL `https://tdl.etsi.org/index.php/downloads`.
12. S. Fabbri, J. Maldonado, and M. Delamaro. Proteum/fsm: a tool to support finite state machine validation based on mutation testing. In *Proceedings. SCCC'99 XIX International Conference of the Chilean Computer Science Society*, pages 96–104, 1999.
13. T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story diagrams: A new graph rewrite language based on the unified modeling language and java. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Theory and Application of Graph Transformations*, pages 296–309. Springer Berlin Heidelberg, 2000.
14. P. Gómez-Abajo, E. Guerra, and J. de Lara. Wodel: A domain-specific language for model mutation. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, SAC '16, page 1968–1973. Association for Computing Machinery, 2016.
15. P. Gómez-Abajo, E. Guerra, J. de Lara, and M. G. Merayo. Wodel-test: a model-based framework for language-independent mutation testing. *Software and Systems Modeling*, 20:1–27, 2020.
16. D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. Statemate: a working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, 1990.
17. N. Hili, M. Bagherzadeh, K. Jahed, and J. Dingel. A model-based architecture for interactive run-time monitoring. *Software and Systems Modeling*, 19:959–981, 2020.
18. J. Iqbal, A. Ashraf, D. Truscan, and I. Porres. Exhaustive simulation and test generation using fuml activity diagrams. In P. Giorgini and B. Weber, editors, *Advanced Information Systems Engineering*, pages 96–110, Cham, 2019a. Springer International Publishing.
19. Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011.
20. F. Khorram, E. Bousse, J.-M. Mottu, and G. Sunyé. Adapting tdl to provide testing support for executable dsls. *Journal of Object Technology*, 20(3):6:1–15, 2021.
21. T. M. King, G. Nunez, D. Santiago, A. Cando, and C. Mack. Legend: An agile dsl toolset for web acceptance testing. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, page 409–412. Association for Computing Machinery, 2014.
22. T. Kos, M. Mernik, and T. Kosar. Test automation of a measurement system using a domain-specific modelling language. *Journal of Systems and Software*, 111:74 – 88, 2016.
23. I. Lazăr, S. Motogna, and B. Pârv. Behaviour-driven development of foundational uml components. *Electronic Notes in Theoretical Computer Science*, 264(1):91–105, 2010. Proceedings of the 7th International Workshop on Formal Engineering approaches to Software Components and Architectures (FESCA 2010).

24. D. Leroy, E. Bousse, M. Wimmer, T. Mayerhofer, B. Combemale, and W. Schwinger. Behavioral interfaces for executable dsls. *Software and Systems Modeling*, 19(4):1015–1043, 2020.
25. J.-h. Li, G.-x. Dai, and H.-h. Li. Mutation analysis for testing finite state machines. In *2009 Second International Symposium on Electronic Commerce and Security*, pages 620–624, 2009.
26. D. Lübke and T. van Lessen. Bpmn-based model-driven testing of service-based processes. In *Enterprise, Business-Process and Information Systems Modeling*, pages 119–133. Springer, 2017.
27. P. Makedonski, G. Adamis, M. Käärik, F. Kristoffersen, M. Carignani, A. Ulrich, and J. Grabowski. Test descriptions with etsi tdl. *Software Quality Journal*, 27 (2):885–917, 2019.
28. T. Mens, A. Decan, and N. I. Spanoudakis. A method for testing and validating executable statechart models. *Software and Systems Modeling*, 18:837–863, 2019.
29. B. Meyers, J. Denil, I. Dávid, and H. Vangheluwe. Automated testing support for reactive domain-specific modelling languages. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, pages 181–194. Association for Computing Machinery, 2016.
30. S. Mijatov, T. Mayerhofer, P. Langer, and G. Kappel. Testing functional requirements in uml activity diagrams. In J. C. Blanchette and N. Kosmatov, editors, *Tests and Proofs*, pages 173–190, Cham, 2015. Springer International Publishing.
31. OASIS. Web services business process execution language version 2.0, 2007.
32. Object Management Group. Business Process Model And Notation, 2010.
33. Object Management Group. Semantics of a Foundational Subset for Executable UML Models, 2013.
34. Object Management Group. Meta Object Facility, 2016.
35. Object Management Group. Unified Modeling Language, 2017.
36. Object Management Group. Precise Semantics of UML State Machines, 2019.
37. S. Pinto Ferraz Fabbri, M. Delamaro, J. Maldonado, and P. Masiero. Mutation analysis testing for finite state machines. In *Proceedings of 1994 IEEE International Symposium on Software Reliability Engineering*, pages 220–229, 1994.
38. D. Santiago, A. Cando, C. Mack, G. Nunez, T. Thomas, and T. M. King. Towards domain-specific testing languages for software-as-a-service. In *2nd International Workshop on Model-Driven Engineering for High Performance and Cloud computing (MDHPCL)*, pages 43–52, 2013.
39. F. Siavashi, D. Truscan, and J. Vain. Vulnerability assessment of web services with model-based mutation testing. In *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 301–312, 2018.
40. D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008.
41. H. Wu, J. Gray, and M. Mernik. Unit testing for domain-specific languages. In W. M. Taha, editor, *Domain-Specific Languages*, pages 125–147. Springer Berlin Heidelberg, 2009.