Introducing CreaTest: a Framework for Test Case Generation in itemis CREATE

Andrea Bombarda[0000−0003−4244−9319], Silvia Bonfanti[0000−0001−9679−4551], Angelo Gargantini[0000−0002−4035−0131], and Nico Pellegrinelli[0009−0000−4944−6845]

University of Bergamo, Bergamo, Italy
{andrea.bombarda, silvia.bonfanti, angelo.gargantini, nico.pellegrinelli}@unibg.it

Abstract. In model-driven engineering, models are used to specify, validate, and verify the system design and generate code and tests. In all these activities, assuring the correctness of the model is paramount, as to derive correct code and tests, a correct model is required. In this paper, we introduce CreaTest, a framework designed to generate abstract tests for itemis CREATE Statecharts by leveraging existing code-based test generators. Our approach consists of translating Statecharts into executable Java code, and then applying a white-box test generation tool, like Evosuite, to produce JUnit test cases. Test cases are subsequently abstracted back into a format supported by the original modeling tool for model validation. We evaluate CreaTest on a large set of Statecharts retrieved from GitHub. Our results show that CreaTest generates high-coverage abstract tests for Statecharts and that optimizing the generated code significantly enhances the effectiveness of test generation.

Keywords: Abstract Tests · Statecharts · EvoSuite · itemis CREATE

1 Introduction

Automatic test generation from models has been a challenge for many years [13, 30]. It is a fundamental aspect of applying model-based testing (MBT) approaches. In the literature, many formal notations for models have been proposed, ranging from Finite State Machines [22] to Abstract State Machines [5] and from Extended Finite State Machines [3] to UML Statecharts [6] and Harel Statecharts [20], to name a few. In this paper, we analyze itemis CREATE,[1], a commercial modular toolkit for developing, simulating, and generating code from a particular type of executable finite-state machines known as CREATE Statecharts, a variant of Harel Statecharts.[2] It has been used in real-world projects [1, 23], where developers have taken advantage of its automatic code generation features and, unlike UML, its precise semantics.

[1] https://www.itemis.com/en/products/itemis-create/

[2] Up to version 3.x, itemis CREATE was named itemis Yakindu and was open source: https://github.com/itemisCREATE/Statecharts

One of the desired features of a formal notation is the possibility of writing and executing abstract tests [12,28]. Such tests can be used to validate the formal model, for regression testing, and for conformance testing of the implementation. However, most formalisms do not provide a feature for automatic abstract test generation. Indeed, given the great number of formalisms and the non-standardization between tools, while test generation from code has made great strides, providing significant results in terms of applicability, research effort, and effectiveness [17], model-based test generation is still not well-supported, and many tools completely lack test generation functionality. This is the case with CREATE: despite its excellent support for test execution and coverage evaluation, it does not provide any mechanism for automatic test generation.

To bridge this gap, in this paper, we propose a framework exploiting test generators for code to generate abstract tests aimed at achieving the desired level of structural coverage. The approach briefly consists of exploiting a code generator that translates the model to code in a target programming language (e.g., Java), applying white-box test generators to obtain test cases in the target language, and, finally, abstracting them back to the formalism supported by the modeling tool. We implement this approach in CreaTest: by starting from the `itemis CREATE` *Statecharts*,we generate Java code thanks to the code generation utility offered by the tool. Then, we use EvoSuite [16] to generate JUnit test cases, and we abstract them into the SCTUnit format supported by `itemis CREATE`. We apply some optimizations during this process, reducing the search space to facilitate the EvoSuite test generation and achieve higher coverage.

We evaluate CreaTest by applying it to a set of 133 Statecharts, with different complexity, gathered from GitHub. With such an extensive set of benchmarks, we are able to assess the effectiveness and the performance of CreaTest and to identify the limitations of our approach. In addition, we evaluate the impact of the code optimizations aimed at helping EvoSuite in test generation on the coverage of the abstract tests. Our results confirm that using the proposed methodology is a viable solution for solving the problem of generating abstract test cases for Statechart validation. Indeed, for most of the models considered, we have been able to cover a high percentage of the models' behavior by simply relying on automatically generated test cases. Moreover, we have found that optimizing the code generated by `itemis CREATE` is effective for significantly increasing the Statecharts coverage and helping EvoSuite to be more effective.

The paper is structured as follows. Sect. 2 discusses the background for the CreaTest approach, while Sect. 3 describes the CreaTest approach and tool.The effectiveness of our approach is evaluated in Sect. 4, while possible threats to validity are discussed in Sect. 5. Finally, Sect. 6 compares our approach with related works, and Sect. 7 concludes the paper.

2 Background

In this section, we present the formalism supported by `itemis CREATE`, i.e., the Statecharts, its code generator, and the language used to write abstract tests.

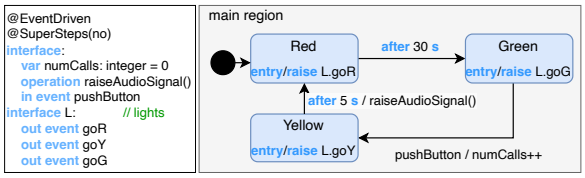

Fig. 1: Example of an `itemis CREATE` Statechart for a simple traffic light

2.1 `itemis CREATE` Statecharts

`itemis CREATE` uses CREATE Statecharts, based on Harel Statecharts and very similar to UML state machines. An example of a Statechart for a simple traffic light is shown in Fig. 1. In the following, we analyze the main CREATE Statecharts features necessary to understand the process we implement in CreaTest.

Two execution schemes exist for CREATE Statecharts: event-driven and cycle-based. In the former, a run-to-completion computation step is executed each time an event is raised, while in the latter, events are collected and then processed periodically in a run-to-completion computation step.

Thanks to the graphical editor, users can define *states* (like `Red` in Fig. 1) and regions (like `main region` in Fig. 1), both of which can also be hierarchical. *Transitions* between states can be defined and are fired when an event happens. The firing of a transition can be limited through *guards*. In addition, the system reaction when a transition occurs can be declared. For example, the transition between `Green` and `Yellow` is fired when the *incoming* event `pushButton` happens, and the system reacts by incrementing the `numCalls` *variable*.

Regardless of the states and transitions, `itemis CREATE` allows for defining the *interface* of the system, representing the entities that are externally visible and used by the client code to interact with the Statechart. A single Statechart can have multiple interfaces, which can be unnamed or named (see the `l` (lights) interface in Fig. 1). In each interface, developers may define `in events`, which are incoming events raised by the client code and processed by the state machine; `out events`, raised by the state machine and delivered to the external environment; `vars`, used to store data; and `operations`, which make an external functionality implemented by the client code accessible by the state machine. In addition, a Statechart can import another one to enable collaboration between them.

2.2 Code generator

`itemis CREATE` supports executable code generation from Statecharts to C++, C, Java, and Python. In this work, we focus on Java code generation, but the same considerations we draw, can be applied to other programming languages. The generation of executable code from the Statechart starts with the definition of the `SGen` model, used to configure the code generation process.

Once the translation rules are defined, `itemis CREATE` generates source code by exploiting several interfaces, such as: 1. `IStatemachine`, which contains the

```
1  public class TrafficLight implements ITimed,
       IEventDriven {
2      public static class L { ... }
3      protected L l;
4      private long numCalls;
5      ...
6      public TrafficLight() { l = new L(); ... } ...
7  }
```

Listing 1: Java class generated for the Statechart TrafficLight

Table 1: Mapping between `itemis CREATE` objects and Java

`itemis CREATE`	Java
Named interface	Inner class
Statechart variables	Fields in the state machine class (with get and set)
Incoming events	Methods
Outgoing events	Observable objects
Operations	Inner interfaces
States	Enumerations

declaration of methods to be implemented by the state machine (e.g., `enter`, `exit`, `isActive`, and `isFinal`); 2. `IEventDriven` or `ICycleBased`, depending on the type of the state machine; and 3. `ITimed` and `ITimerService`, generated when at least a timed incoming event is present, and containing the declaration of methods useful when the Statechart contains timers.

Then, `itemis CREATE` defines a single class for the Statechart, as shown in Listing 1 for the Statechart in Fig. 1. Each named interface defined by the Statechart is translated into an inner class (e.g., line 2), while all fields and methods in an unnamed interface, if any, are defined directly in the state machine class (e.g., line 4). Further mappings between `itemis CREATE` objects and Java constructs are shown in Tab. 1.

2.3 SCTUnit tests

In the Model-Driven Engineering approaches, assuring the correctness of the model, i.e., it correctly captures the requirements, is of paramount importance. Having a faulty model jeopardizes all the analysis activities (e.g., simulation) and defeats the purpose of deriving artifacts from it (e.g., the implementations).

To validate the machine, `itemis CREATE` offers the possibility to write SCTUnit tests.[3] Each test consists of a well-defined sequence of instructions that are applied to the state machine under test and possibly change that state machine in a specific way. The user specifies in a test the expected effects of these changes as assertions, which are automatically checked by `itemis CREATE`. SCTUnit tests are of paramount importance when developing a Statechart. They can be used to support the Test-Driven Development process, conformance testing to verify the compliance of the model behavior with that specified in the requirements document, and regression testing. Furthermore, they can be concretized during MBT activities to obtain unit tests for the implementation.

Test cases must be defined within a class contained in a `.sctunit` file. The typical structure of a test case starts with an `enter` statement followed by a sequence of `raise`, `proceed`, and `assert`, as shown in the example in Listing 2 for the Statechart in Fig. 1. The `enter` statement initializes and starts the state machine. The `raise` statements are used to raise incoming events. The `proceed`

[3] `https://tinyurl.com/45wsvmm2`

```
1  testclass TLTest for Statechart TrafficLight {
2      @Test operation workingStateTest() {
3          enter
4          assert active(main_region.Red)
5          proceed 35 s
6          assert active(main_region.Green)
7          raise pushButton
8          assert active(main_region.Yellow)
9          exit
10     } ... }
```

Listing 2: An example of SCTUnit test class

statements are used to make the state machine proceed for a given amount of time or cycles. Finally, the `assert` statements are used to check a specific condition of the system, e.g., whether a state is active or not, an operation has been called or not, an outgoing event has been raised or not. In addition, SCTUnit provides features such as, but not limited to, conditional statements (`if` and `if else`), loops (`while`), and operation mocking.

SCTUnit tests can be automatically translated into different languages (C, C++, Java, and Python), each one with its own testing framework. However, no backward translation (i.e., from the target language to the SCTUnit format) is available and, thus, concrete test cases cannot be used to derive SCTUnit tests.

3 CreaTest approach for test generation

In this section, we present CreaTest, our test generator for `itemis CREATE` Statecharts, and we explain it based on the example shown in Fig. 1. First, we describe the basic process we implemented and discuss possible optimizations to be applied to the generated source code. Then, we present how we automatically gather the information used for test translation. Finally, we describe the process we follow for translating the tests automatically generated into SCTUnit tests. The CreaTest tool is available in our replication package [10].

3.1 The CreaTest basic process

The UML activity diagram shown in Fig. 2 models the CreaTest process.

Initially, CreaTest takes as input an `itemis CREATE` Statechart, as described in Sect. 2.1, and automatically builds the `SGen` model, which contains the properties and settings for code generation. In this step, CreaTest extracts the name from the Statechart, which is a mandatory input parameter. Having defined the properties for the code generation process, CreaTest exploits the `itemis CREATE` Java code generator to generate a Java file containing the class implementing the input Statechart. This procedure requires as input the Statechart and the `SGen` model. Then, the obtained Java class is transformed into a *optimized* one to help the EvoSuite test generator. We describe this step in more detail in Sect. 3.2.

At this point, the real test generation process can start. The Java code of the Statechart is given as input to EvoSuite [16], which generates JUnit test cases by exploiting evolutionary algorithms. Finally, JUnit tests are abstracted and translated into SCTUnit tests.During this process, additional information is

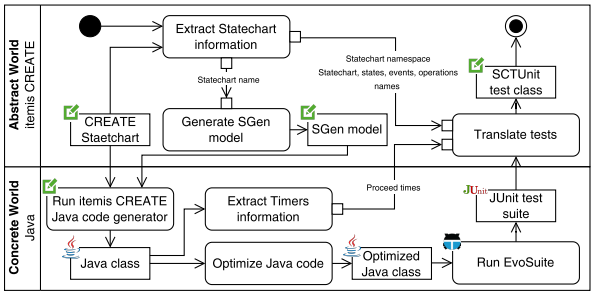

Fig. 2: CreaTest process

required. In particular, the translation needs the Statechart namespace and the Statechart, states, events, operations, and interface names, which are retrieved directly from the model, and information about the proceed times of timed events, which is retrieved from the generated Java class.

In the following subsections, we explain in more detail how the optimization of the Java source code is performed by CreaTest, how additional information is collected, and how the translation of JUnit into SCTUnit works.

3.2 Helping EvoSuite by hacking the generated code

EvoSuite can leverage different search algorithms and aims to maximize a set of coverage criteria. In this work, DynaMOSA [24] is used as the search algorithm and the selected coverage criteria are branch, methodnoexception, and output.[4] For the tool to generate a test suite, the greater the search space, the higher the generation time and, possibly, the lower the coverage reached. For this reason, in CreaTest we have implemented strategies aimed at optimizing the Java source code generated by the `itemis CREATE` Java code generator for test generation. These optimizations focus on reducing the search space by "hiding" all details that are not meaningful for the Statechart and should not be used by EvoSuite. Briefly, the optimizations introduced by CreaTest modify the visibility of members that should not be used in the test cases, e.g., because there is no corresponding expression or statement in SCTUnit for that member. Considering that EvoSuite tries to interact with *public* and *protected* members, every member that is not of interest for the Statechart should be forced to be *private* in the optimized Java class. The optimizations performed by CreaTest (see Tab. 2) relate both to fields and methods. As a general rule, since our goal is to achieve maximum Statechart coverage, we set all fields or methods that would not contribute to increasing coverage to *private*. In particular, both *protected* and *public* fields are converted to *private* ones, as they should not be directly accessed. All *protected* methods (associated with internal or outgoing events, operations, or setting the enabling status for a state) are converted into *private* ones, as they have no corresponding statements in SCTUnit.

[4] Further details on the EvoSuite configuration within CreaTest can be found in [10].

Table 2: Optimizations of Java members

Original Java	Optimized Java
protected methods	private methods
protected fields	private fields
public fields	private fields
public methods (interfaces)	public methods (interfaces)
set methods	private set methods
get methods	private get methods

```
1  @Test(timeout=4000) public void test10() throws Throwable {
2      TrafficLightS TrafficLightS0 = new TrafficLightS();
3      ITimerService iTimerService0 = mock(ITimerService.class, new ViolatedAssumptionAnswer());
4      TrafficLightS0.setTimerService(iTimerService0);
5      TrafficLightS0.enter();
6      TrafficLightS0.raiseTimeEvent(0);
7      TrafficLightS0.State TrafficLightS_St0 = TrafficLightS0.State.MAIN_REGION_GREEN;
8      boolean boolean0 = TrafficLightS0.isStateActive(TrafficLightS_St0);
9      assertTrue(boolean0); }
```

Listing 3: JUnit test case

The methods defined in an interface are kept *public* to avoid compilation errors. *Set* methods are made *private* to prevent EvoSuite from modifying the values of variables without executing an action on the Statechart. Finally, *get* methods are forced as private. During code optimization, additional methods may be added to make EvoSuite test generation more effective. This is the case with cycle-based Statecharts. When `itemis CREATE` generates the Java code, a `runCycle()` method is provided to perform one computation step. However, EvoSuite tries to keep the generated test cases as short as possible, and this implies that behaviors shown only after many steps may be untested. For this reason, during the code optimization phase, we add the `proceedCycles(int nCycles)` method, allowing EvoSuite to perform more cycles without having to call the already existing `runCycle()` method multiple times.

An example of a JUnit test generated by EvoSuite for the Statechart reported in Fig. 1 after the optimization of the Java code is shown in Listing 3.

3.3 Collecting Statechart and Timers information

For the JUnit tests to be translated into SCTUnit tests, it is necessary to collect additional information from the Java source code and the CREATE Statechart. While a simple parser suffices for extracting information from Java, analyzing the Statechart requires de-serializing its XML.

Concerning the Java code, CreaTest extracts the mapping between temporal event IDs and the associated proceed time (used by the *after* and *every* statements, as in the example shown in Fig. 1). If a timed event is present in the Statechart under analysis, a `setTimer` statement will be present in the Java code. This is used by the state machine to start a timer for a given `eventID` and lasts for a given `time`. Furthermore, the expiration of a timer identified by

```
1  <sgraph:Statechart [...] specification="@EventDriven..." name="TrafficLight">
2      <regions [...] name="main region"> [...] <vertices xsi:type="sgraph:State" [...] name="Red">
3          <outgoingTransitions [...] specification="after 30 s"/>
4      </vertices> [...] </regions> </sgraph:Statechart>
```

Listing 4: Excerpt of the `.ysc` of the Statechart in Fig. 1

an `eventID` is handled by the `raiseTimeEvent` method (see line 6 in Listing 3). Thus, for the JUnit code to be translated into the corresponding SCTUnit, the link between a defined `eventID` and the corresponding duration must be found. To do this, the Java code is parsed and all `setTimer` statements are analyzed to extract the link between the `eventID` and the corresponding `time`.

In terms of the Statechart analysis, we have found that the `.ysc` file in which it is saved is in the XMI format, as shown in the excerpt reported in Listing 4, and can be parsed by a classical XML parser. The useful information for the CreaTest approach is that referring to states (and regions), events, and interfaces, which is located in the only node in the XMI with the `sgraph:Statechart` tag. From the attributes of this node it is possible to extract the name of the Statechart, its namespace, input events, interfaces, and operations. From its children, and once the structure of the XMI representation of the Statechart is known, we can extract the hierarchical name of each state.

During code generation, some information, such as the capitalization of names used in the Statechart, is lost. Thus, it is not possible to obtain an SCTUnit class that is correct with respect to the Statechart by looking only at the generated JUnit test cases. For instance, an enum constant `MAIN_REGION_STATEA` could originate from a state in the "main region" named either `StateA` or `STATEA`, with such a distinction being lost during code generation. Similarly, a method such as `raiseMyEvent()` could derive from an event named `myEvent` or `MyEvent`. This issue is addressed by CreaTest, which parses the `.ysc` file to recover the original names and establishes a unique mapping between the generated Java code and the CREATE elements within a single Statechart.

3.4 From concrete to abstract test cases

The last step of the CreaTest process is the translation of concrete test cases into abstract ones, i.e., from JUnit tests generated by EvoSuite to an SCTUnit test executable by `itemis CREATE`. After having performed the steps discussed before, namely the optimization of the Java code and the collection of information from Java classes and the Statechart, the final translation step is straightforward. For the translation of test cases, the input test class is parsed and each method (i.e., test case) is analyzed: an output test case is populated, translating the method calls into SCTUnit statements in the order in which they appear.

Tab. 3 shows the mapping between method calls and SCTUnit statements. Note that the object whose method is called is omitted. All methods, except for `doReturn` mock methods, must be called from an instance that implements the

Table 3: Mapping between JUnit and SCTUnit statements (`myInt.x` is used to indicate that the CREATE element `x` is defined by the interface `myInt`)

JUnit	**SCTUnit**
`enter()`	`enter`
`exit()`	`exit`
`triggerWithoutEvent()`	`triggerWithoutEvent`
`runCycle()`	`proceed 1 cycle`
`proceedCycles(`X`)`	`proceed` X `cycle`
`raiseTimeEvent(0)`	`proceed` X`ms`
`raiseMyEvent()`	`raise myEvent / raise myInt.myEvent`
`raiseMyTypedEvent(`X`)`	`raise myTypedEvent:` X `/ raise myInt.myTypedEvent:` X
`doReturn(`X`).when().myOp()`	`mock myOp returns (`X`) / mock myInt.myOp returns (`X`)`
`assertTrue(`$expr$`) / assertFalse(`$expr$`)`	`assert` $expr$ `/ assert !`$expr$

```
1  @Test operation test10() {
2      enter
3      proceed 30s
4      assert active(TrafficLight.main_region.Green) }
```

Listing 5: SCTUnit test case

Statechart or from an instance of its inner classes, where stated, which are the interfaces. The possible interface to be used for an operation is obtained from the argument of the `when` method call that follows the `doReturn` call. A particular translation is that concerning the `assertTrue` and `assertFalse` when an *expr* is used. This can be a call of the `isActive` method (translated into the `is_active` SCTUnit statement), of the `isFinal` method (translated into the `is_final` SCTUnit statement), or of the `isStateActive(Statechart.State.REGION_STATE)` method (translated into the `active(Statechart.region.state)` SCTUnit statement). Finally, considering that SCTUnit test cases interact with the Statechart only by raising events or calling methods, the translation process only translates method calls and ignores everything else.

By applying such a process, the JUnit test case reported in Listing 3 is translated into the SCTUnit test shown in Listing 5. Note that during this translation, the information collected in the process described in Sect. 3.3 has been used (e.g., the duration of the timer and the name of the states).

4 Experiments

In this section, we describe the experiments we have performed to evaluate CreaTest, its performance, and its limitations. More specifically, the experiments have been designed to answer the following research questions:

RQ1 How effective is CreaTest at generating correct test classes while achieving high coverage?

RQ2 Does providing optimized input to the test generator enhance the quality of the generated SCTUnit test classes?

RQ3 Does the coverage achieved by EvoSuite relates to the coverage provided by the final SCTUnit test class?

In the following, we describe the experimental methodology we have used to gather and analyze data to answer the presented RQs.

4.1 Experimental methodology

In the following, we delve into the experimental methodology used in our experiments. First, we analyze the models we have selected as benchmarks and how they have been filtered to keep only the meaningful ones. Then, we describe the test generation process and how we gathered the data used to address our RQs.

Benchmark collection: In May 2024, we gathered 219 CREATE Statecharts from GitHub, with many of them coming from two key repositories: the official `itemis CREATE` repositories [2] and the repository [25] of STL4IoT, a library of Statechart templates for designing IoT systems composed of atomic CREATE Statechart components. We retrieved 68 Statecharts from the former, 54 from the latter, and the remaining 97 from other repositories.

Benchmark inspection: Before running CreaTest on all benchmarks, a manual inspection of each Statechart was performed to remove Statecharts with errors or those not suitable as input for the tool. In total, we removed 86 Statecharts (48 duplicates, 16 importing C/C++ code, 1 using the SCXML domain, 9 with errors, 15 too simple or meaningless). In the end, we selected 133 Statecharts for in our experiments. For each of them, we measured the number of states and their average and maximum depth.[5]These Statecharts are available in our replication package [10].

Test generation and execution: For each benchmark, we generated an abstract test suite with CreaTest both enabling and disabling the Java code optimization. This process was automated by the batch script available in our replication package. The resulting test suites were manually checked to determine whether any of them presented static errors (e.g., syntax errors). Finally, by executing the SCTUnit test suites on the Statecharts, we gathered all coverage information for both standard and optimized tests. More specifically, when discussing the EvoSuite coverage, we considered the average of the coverage criteria used when executing EvoSuite (i.e. branch, output, methodnoexception), while in the case of `itemis CREATE` the considered coverage is the average coverage of *external* regions (i.e., the average coverage of the states in them, computed, for each state, as the ratio of exiting transitions executed by the tests w.r.t. their total number). The generated SCTUnit test suites and the coverage data are available in the replication package [10].

4.2 RQ1: Effectiveness of CreaTest

In this RQ, we are interested in investigating whether CreaTest is effective, i.e., it can generate tests with no static errors (syntactically *correct*) and that

[5] The depth of a state refers to its level within the hierarchical structure of the Statechart.

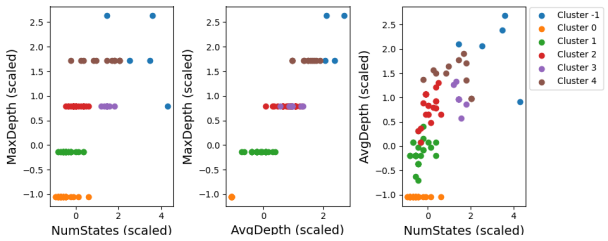

Fig. 3: Two-dimensional views of DBSCAN clustering scatter plot.

pass (semantically *correct*). Furthermore, we are interested in evaluating the coverage achieved by the generated test cases considering models of different complexities. For each of the 133 Statecharts in our test set, the tool was able to generate one SCTUnit test class. None of the generated test classes contained compilation errors, and only 1 test class contained at least a failing test. Thus, out of the 133 Statecharts, for 132 of them it was possible to generate a correct test suite, leading to a success rate of 99.25%. In 3 cases, the generated tests allowed us to discover a faulty behavior of the Statechart: the test execution halted due to the presence of a loop in the Statechart, indicating that certain executions result in infinite cycles. Except for halting executions, coverage was collected for the remaining 130 test classes, resulting in an average coverage of 60.05%.

We are also interested in investigating the impact of Statechart complexity on the coverage of the generated test cases. For this purpose, we grouped the Statecharts into clusters. To define the clusters, we decided to include the following features: number of states in the Statechart, average state depth, and maximum state depth. Furthermore, we used DBSCAN as a clustering algorithm, with its implementation in Python provided by the library scikit-learn. Fig. 3 shows the graphical 2-D representation of the clusters we found using the three features mentioned above. As reported in the figure, five clusters and five outliers (cluster -1) were identified. It is apparent that the main feature used to discriminate clusters is the maximum depth of the states, which is sufficient to uniquely identify clusters 0, 1, and 4. Clusters 2 and 3 have the same value of maximum depth for all their samples and can be discriminated by the number of states. The average depth was found to be the least useful feature for clustering. In Tab. 4, we detail the characteristics of each cluster. More specifically, we report the characteristics of each cluster, namely the number of Statecharts in the cluster, the average number of states across the Statecharts in the cluster, and the common maximum depth. We also report the results achieved by CreaTest, namely the average coverage and the number of Statecharts for which the coverage is 0%, between 1% and 49%, between 50% and 99%, and exactly 100%. Note that the cluster -1 is the set of outliers; thus, we do not report the maximum depth. By analyzing the results we obtained in each different cluster we can discuss the impact of Statechart complexity on the performance of CreaTest. First, even in cluster 0, the simplest one, we have 4 Statecharts with null coverage (i.e. the

Table 4: Characteristics of all clusters

Cluster	#	Max Depth	Avg. States	Avg Cov.	#0%	#1-49%	#50-99%	#100%
0	48	1	3.60	75.8%	4	5	19	20
1	33	2	5.39	67.1%	4	3	20	6
2	23	3	9.30	54.6%	2	6	15	0
3	9	3	21.11	23.8%	5	1	3	0
4	12	4	19.42	33.1%	3	5	4	0
-1	5	-	34.60	17.4%	1	3	1	0

Table 5: Comparison of SCTUnit classes in standard and optimized Java code.

Metric	Standard code	Optimized code
# of generated classes	133	133
# of classes with passing tests	101	129
# of classes with failing tests	27	1
# of classes with errors	4	0
# of classes with blocking tests	1	3
Average coverage	19.45%	60.05%

tests do not contain enter statements). Moreover, in each cluster, we obtained some empty or meaningless test cases (e.g., with a single statement). This shows that there are Statecharts with few states for which CreaTest struggle to generate tests. However, the tool performance seems to be strongly correlated with the number of states in the Statechart. Indeed, from cluster 0 to -1 (the one containing the most complex Statecharts considered in our analysis) the average coverage decreases and, by comparing clusters 3 and 4, CreaTest performs better on the latter than with the former, as it presents fewer states.

Our experiments show that CreaTest is able to generate correct and passing SCTUnit test cases for more than 99% of the considered `itemis CREATE` Statecharts. As expected, the lower the model complexity, the higher the coverage level of the generated tests.

RQ1: CreaTest **effectivness**

4.3 RQ2: Impact of Java code optimization on tests

For this RQ, we analyzed the impact of using code optimization on the Java code produced by `itemis CREATE` and used by EvoSuite for generating test cases. As shown in Tab. 5, our experiments confirm that using the *optimized* code reduces the search space of EvoSuite and allows the tool to find better solutions, reaching higher coverage. Additionally, the experiments show that this reduction in the search space allows increasing the number of Statecharts for which the tool is able to generate a passing SCTUnit test class and leads to an overall improvement in the quality of the generated SCTUnit tests, both in terms of coverage and static errors. Indeed, by optimizing the source code, we prevent Evosuite from inserting Java statements not having direct correspondence with SCTUnit ones, and this allows EvoSuite to focus only on relevant statements.

Our results show that optimizing Java source codeallows to cover up to 40.6% more of the Statecharts and increases the percentage of passing tests.

RQ2: Impact of code optimization

4.4 RQ3: Correlation between JUnit and SCTUnit coverage

EvoSuite aims to generate a JUnit test suite for the Java class given as input with the highest possible coverage. In the context of the CreaTest approach, the JUnit test suite generated by EvoSuite is only an intermediate result. However, it would be beneficial if the effort of EvoSuite also relates to the quality of the final results. In other words, if the coverage of the JUnit test suite generated by EvoSuite is high, the coverage of the final SCTUnit test case should also be high. Thus, in our experiments, we measured the Pearson correlation between the two coverages, i.e., the one obtained by EvoSuite on the Java code and the one of the SCTUnit test cases on the CREATE Statecharts. The resulting correlation is 0.840, i.e., the two measures are strongly correlated.

The correlation between JUnit and SCTUnit test cases is 0.840: this means that EvoSuite can cover most of the Java code; then, with CreaTest, we can transfer this good result to the Statecharts as well.

RQ3: Correlation of JUnit with SCTUnit coverage

5 Threats to validity

In this section, we discuss potential threats to the validity [26] of our work and the actions we have taken to mitigate them.

Internal validity refers to the fact that the different outcomes obtained with the analyzed techniques and tools are actually caused by the different approaches themselves and by the way the experiments were carried out, and not by methodological errors. To mitigate this risk, we have carefully checked the code executed in our experiments to see if there could be other factors that may have caused the outcome, such as errors in the tools or in the experimental code we wrote.

A possible threat to the *construct validity* comes from the assumption that the measures we considered for evaluating the CreaTest approach (i.e., the executability of the test cases, the pass/fail ratio, and the coverage) are suitable to assess our approach correctly. We rely on the literature for this, where similar measures are often used [4, 21, 31].

External validity is concerned with whether we can generalize the results outside the scope of the presented benchmarks and tool. Concerning the benchmarks used, we have tried to gather all the `itemis CREATE` Statecharts we found on GitHub and we believe they are representative of the domain. One may argue that the applicability of the approach of generating abstract test cases with test generators for code is limited only to the CREATE tool. However, we think that the same approach can be extended to any other formal notation and tool where the tool allows for generating executable code from the model and the tool has its

own validation language. What we presented in this paper is a proof-of-concept and, as future work, we may investigate its generalizability and applicability to other formalisms, such as Matlab Simulink or ASMETA [5]. The validity of the proposed approach depends on the correctness of the source code generator: our assumption is that the behavior implemented by the Java source code corresponds to that of the Statechart. We did not find any issue in the code generation, `itemis CREATE` is a reliable software, and we can conjecture that the code generator is well-developed and tested.

Since there are no tools for generating tests from CREATE Statecharts, we could not compare CreaTest with other approaches. We suspect that an ideal tool avoiding translation to Java code and operating directly at the model level would be more efficient. However, writing such a tool would require a great effort – as also suggested by the fact that such a tool does not exist. We have carried out some experiments with EvoMBT [15], but they were inconclusive: the translation from CREATE Statecharts to EvoMBT is complex and hard to automate, jeopardizing any possible advantage in its usage for this approach.

6 Related Work

In this section, we report a non-exhaustive summary of other approaches and formalisms used in the literature to perform MBT and generate abstract tests, and we highlight the differences with the approach we propose in this paper.

State machines (in one of their variants) are the most adopted formalism when it comes to MBT. In [8], Abstract State Machines, with the ASMETA [9] framework, are used as a modeling notation and abstract test cases are automatically generated by exploiting the counterexample provided by the NuSMV model checker [11]. The user must then provide a concretization mechanism to obtain concrete test cases runnable on the system under test. Despite the completeness of the ASMETA framework, its use is not intuitive and the models have to be written in a textual format, whereas with `itemis CREATE` users can take advantage of a graphical and intuitive representation. Abstract State Machines are also exploited as an underlying formalism by Spec Explorer [27]. As for ASMETA, there is no graphical editor, and the model programs can be defined using the C# programming language or any other .NET language, introducing higher complexity in the modeling phase. Other works have proposed the use of Extended Finite-State Machines (EFSMs) to model the system. For example, EvoMBT [15] allows the user to implement an EFSM in Java by using the provided interfaces. Then, once the user has modeled the SUT (or part of it) as an EFSM, EvoMBT can generate abstract tests by exploiting the same tool we propose in this paper, namely EvoSuite. The user must provide a concretization mechanism for the generation of executable test cases runnable on the SUT. A different approach for modeling a system and deriving test cases is the one exploited by GraphWalker [29]. This requires the user to model the system as a directed graph, where edges represent actions on the SUT and nodes represent

verifications (i.e., assertions on the SUT). GraphWalker produces abstract tests as paths (lists of pairs of edges and vertices) on the graph.

Regarding `itemis CREATE`, to the best of our knowledge, no ready-to-use tool exists. The only approach we found is Y2U [18], in which the authors propose to automatically translate Yakindu Statecharts (thus, the old version of the tool we analyze in this paper) into UPPAAL [7] timed automata. The test generation process has not been implemented, but we believe that tests may be automatically derived thanks to the UPPAAL model checker. Exploiting the combination of a code generator and a test generator for code is also presented in [14]. However, in that paper, the emphasis is more on code generation (from PLC to Python). Our approach's strength is to leverage code generators already provided by the modeling tool, such as `itemis CREATE`.

7 Conclusion

Abstract test generation from formal models is a crucial aspect of modeling formalisms. These tests enable users to validate models and conduct regression and conformance testing once the actual implementation is available. In this paper, we have introduced CreaTest, marking the first attempt to develop an abstract test generator for the `itemis CREATE` Statechart formalism. Our approach leverages the code generation capabilities of `itemis CREATE` to produce Java code, so avoiding the otherwise cumbersome task of providing a translation from Statecharts to other tools (such as model checkers or SMT solvers). We then generate JUnit test cases by using EvoSuite, which we subsequently abstract back into the language supported by `itemis CREATE`, SCTUnit. Before using EvoSuite, we also optimized the code to facilitate generation. With our experiments, we have evaluated CreaTest. Moreover, we have shown the positive impact of code optimization and the capability of CreaTest to generate abstract test cases, achieving high coverage on the Statecharts. This work is a proof of concept and, as future work, we may investigate the generalization of the same approach to other modeling formalisms and reducing the code optimizations (e.g., keeping the `get` methods as public) to not only achieve the state and transition coverage but other coverage metrics as well. Finally, we plan to broaden our evaluation by applying more suitable complexity metrics to assess the impact of complexity on coverage, including cyclomatic complexity and its variants for hierarchical Statecharts proposed in [19].

Acknowledgments. This work is supported by PNC - ANTHEM (AdvaNced Technologies for Human-centrEd Medicine) - Grant PNC0000003 – CUP: B53C22006700001 - Spoke 1 and by SERICS (PE00000014) under the NRRP MUR program funded by the EU - NGEU and by the European Union - Next Generation EU. We also acknowledge financial support of the project SAFEST (Trust assurance of Digital Twins for medical cyber-physical systems), funded by the European Union - Next Generation EU, Mission 4, Component 2, Investment 1.1, CUP F53D23004230006, under the National Recovery and Resilience Plan (NRRP) – Grant Assignment Decree No. 959 adopted on 30 June 2023 by the Italian Ministry of University and Research (MUR).

References

1. Abba, A., et al.: The novel Mechanical Ventilator Milano for the COVID-19 pandemic. Physics of Fluids **33**(3) (Mar 2021). https://doi.org/10.1063/5.0044445
2. itemis AG: itemis CREATE. https://github.com/itemisCREATE (2024)
3. Alagar, V.S., et al.: Extended Finite State Machine, pp. 105–128. Springer London, London (2011). https://doi.org/10.1007/978-0-85729-277-3_7
4. Almaghairbe, R., Roper, M.: Separating passing and failing test executions by clustering anomalies. Software Quality Journal **25**(3), 803–840 (Oct 2016). https://doi.org/10.1007/s11219-016-9339-1
5. Arcaini, P., Bombarda, A., et al.: The ASMETA Approach to Safety Assurance of Software Systems, pp. 215–238. Springer International Publishing, Cham (2021). https://doi.org/10.1007/978-3-030-76020-5_13
6. von der Beeck, M.: Formalization of uml-statecharts. In: UML 2001 — The Unified Modeling Language. Modeling Languages, Concepts, and Tools. pp. 406–421. Springer Berlin Heidelberg (2001). https://doi.org/10.1007/3-540-45441-1_30
7. Bengtsson, J., Larsen, K., Larsson, F., Pettersson, P., Yi, W.: UPPAAL — a tool suite for automatic verification of real-time systems, pp. 232–243. Springer Berlin Heidelberg, Berlin, Heidelberg (1996). https://doi.org/10.1007/BFb0020949
8. Bombarda, A., Bonfanti, S., Gargantini, A.: Automatic Test Generation with ASMETA for the Mechanical Ventilator Milano Controller. In: Testing Software and Systems. pp. 65–72. Springer (2022). https://doi.org/10.1007/978-3-031-04673-5_5
9. Bombarda, A., Bonfanti, S., et al.: Asmeta tool set for rigorous system design. In: Formal Methods. pp. 492–517. Springer Nature Switzerland, Cham (2025). https://doi.org/10.1007/978-3-031-71177-0_28
10. Bombarda, A., Gargantini, A., Pellegrinelli, N.: Replication package for the paper "Introducing CreaTest: a framework for test case generation in itemis CREATE". https://github.com/foselab/CREATest (Apr 2025)
11. Cimatti, A., Clarke, E., et al.: NuSMV 2: An OpenSource Tool for Symbolic Model Checking, pp. 359–364. Springer Berlin Heidelberg (2002)
12. Devroey, X., Perrouin, G., Schobbens, P.: Abstract test case generation for behavioural testing of software product lines. In: Proceedings of the 18th International Software Product Line Conference: Companion Volume for Workshops, Demonstrations and Tools - Volume 2. p. 86–93. New York, NY, USA (2014). https://doi.org/10.1145/2647908.2655971
13. Dias Neto, A.C., et al.: A survey on model-based testing approaches: a systematic review. In: Proceedings of the 1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies. p. 31–36. ACM, New York, NY, USA (2007). https://doi.org/10.1145/1353673.1353681
14. Ebrahimi Salari, M., Enoiu, E.P., Afzal, W., Seceleanu, C.: PyLC: A Framework for Transforming and Validating PLC Software using Python and Pynguin Test Generator. In: Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing. p. 1476–1485. SAC '23, ACM, New York, NY, USA (2023). https://doi.org/10.1145/3555776.3577698
15. Ferdous, R., Hung, C., Kifetew, F., Prandi, D., Susi, A.: EvoMBT: Evolutionary model based testing. Science of Computer Programming **227**, 102942 (2023). https://doi.org/10.1016/j.scico.2023.102942
16. Fraser, G., Arcuri, A.: EvoSuite: automatic test suite generation for object-oriented software. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th

European Conference on Foundations of Software Engineering. p. 416–419. ESEC/FSE '11, ACM, New York, NY, USA (2011). https://doi.org/10.1145/2025113.2025179
17. Gambi, A., Jahangirova, G., Riccio, V., Zampetti, F.: SBST tool competition 2022. In: Proceedings of the 15th Workshop on Search-Based Software Testing. p. 25–32. SBST '22, ACM (2023). https://doi.org/10.1145/3526072.3527538
18. Guo, C., Ren, S., et al.: Transforming Medical Best Practice Guidelines to Executable and Verifiable Statechart Models. In: 2016 ACM/IEEE 7th International Conference on Cyber-Physical Systems (ICCPS). pp. 1–10 (2016). https://doi.org/10.1109/ICCPS.2016.7479121
19. Hall, M.: Complexity metrics for hierarchical state machines. In: Search Based Software Engineering. pp. 76–81. Springer Berlin Heidelberg, Berlin, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23716-4_10
20. Harel, D.: Statecharts: a visual formalism for complex systems. Science of Computer Programming **8**(3), 231–274 (1987). https://doi.org/https://doi.org/10.1016/0167-6423(87)90035-9
21. Lazic, L., Mastorakis, N.: Cost effective software test metrics. W. Trans. on Comp. **7**(6), 599–619 (Jun 2008)
22. Lee, D., Yannakakis, M.: Testing finite-state machines: state identification and verification. IEEE Transactions on Computers **43**(3), 306–320 (1994). https://doi.org/10.1109/12.272431
23. Mohamed, M.A., Kardas, G., Challenger, M.: A Systematic Literature Review on Model-driven Engineering for Cyber-Physical Systems (2021). https://doi.org/10.48550/ARXIV.2103.08644
24. Panichella, A., Kifetew, F.M., Tonella, P.: Automated Test Case Generation as a Many-Objective Optimisation Problem with Dynamic Selection of the Targets. IEEE Transactions on Software Engineering **44**(2), 122–158 (2018). https://doi.org/10.1109/TSE.2017.2663435
25. Rempillo, C., Mustafiz, S.: STL4IoT: A statechart template library for IoT system design (2023)
26. Runeson, P., Höst, M.: Guidelines for conducting and reporting case study research in software engineering. Empirical Softw. Engg. **14**(2), 131–164 (Apr 2009). https://doi.org/10.1007/s10664-008-9102-8
27. Sarma, M., Murthy, P.V.R., Jell, S., Ulrich, A.: Model-based testing in industry: a case study with two MBT tools. In: Proceedings of the 5th Workshop on Automation of Software Test. p. 87–90. AST '10, ACM, New York, NY, USA (2010). https://doi.org/10.1145/1808266.1808279
28. Sprenkle, S., Pollock, L., Simko, L.: A Study of Usage-Based Navigation Models and Generated Abstract Test Cases for Web Applications. In: 4th IEEE International Conference on Software Testing, Verification and Validation. pp. 230–239 (2011). https://doi.org/10.1109/ICST.2011.34
29. Zafar, M.N., Afzal, W., et al.: Model-Based Testing in Practice: An Industrial Case Study using GraphWalker. In: Proceedings of the 14th Innovations in Software Engineering Conference. ISEC '21, ACM, New York, NY, USA (2021). https://doi.org/10.1145/3452383.3452388
30. Zelenov, S.V., Silakov, D.V., et al.: Automatic Test Generation for Model-Based Code Generators. In: Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation. pp. 75–81 (2006). https://doi.org/10.1109/ISoLA.2006.70
31. Zhu, H., et al.: Software unit test coverage and adequacy. ACM Comput. Surv. **29**(4), 366–427 (Dec 1997). https://doi.org/10.1145/267580.267590