UNIVERSITÀ DEGLI STUDI DI BERGAMO

Department of Management, Information and Production Engineering

Master Degree in Computer Engineering

Class LM-32

Automatic Generation of Test Cases for Statecharts in itemis CREATE

Advisor

Prof. Angelo Gargantini

Co-advisor

Dr. Andrea Bombarda

Master Thesis

Nico Pellegrinelli

Student number 1065869

ACADEMIC YEAR 2023/2024

Abstract

In software engineering, leveraging abstract models to guide system design, development and testing is a well-established practice. Model-Driven Development (MDD) and Model-Based Testing (MBT) are two of the most widely utilized methodologies that rely on models. Model-Driven Development (MDD) focuses on creating abstract representations of the system to guide the development process. These models are used to derive executable code. Model-Based Testing (MBT), on the other hand, involves abstracting the System Under Test (SUT), or parts of it, into a model that encodes its intended behavior. From this model, an abstract test suite is derived, representing various scenarios and behaviors to be tested. Subsequently, a concrete test suite, executable over the SUT, is generated from the abstract test suite. In both MDD and MBT, the automatic generation of abstract test cases for a model is a critical step. In MDD, it is important to validate the model to avoid the errors to propagate to the generated code, one way to perform validation is by testing. Manual testing can be time-consuming, error-prone, and inefficient, therefore the testing phase should be automatized and the test cases automatically generated. In MBT, the automatic generation of test cases can be used both for validating the model of the SUT before starting the actual MBT process or for the intermediate step of generating the abstract test cases. The aim of this thesis is to present CREATest and the novel approach it implements. CREATest is a tool specifically designed for the automatic generation of test cases for statecharts in itemis CREATE, formerly known as YAKINDU Statechart Tools. The novel approach for the generation of abstract test cases comprises three fundamental steps: first, the translation of the model into source code; second, the generation of test cases for the source code by exploiting a ready-to-use tool; and third, the abstraction of the generated test cases back to the level of the model.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

This thesis presents a new approach for the generation of abstract test cases for models, as well as the development of a tool that implement this approach for the itemis CREATE tool-kit [19]. Given that the primary application of existing tools for generating abstract test cases is Model-Based Testing (MBT), this thesis focuses on MBT. However, it also explains how an automatic abstract test case generator can be beneficial for Model-Driven Development (MDD).

In the field of software engineering, ensuring the quality and reliability of the product is fundamental, specially in safety-critical domains. Software testing is an approach that allows to systematically identify defects and verify that the system under test (SUT) meets specified requirements. Therefore, testing can improve software quality, increase customer satisfaction and reduce maintenance time.

Given the importance of software testing in the software life-cycle, a lot of techniques and approaches have been proposed to make it more efficient and effective. For instance, one of the most famous methodologies is the agile technique known as Test-Driven Development (TDD). Given the impact and importance of testing in the software life-cycle, TDD puts the emphasis of the whole development process on tests. In TDD, tests, that initially fail, are written before the relative code. Given a failing test, it is implemented just enough source code to make it pass and, then, new test cases are written before implementing new functionalities, and the process restarts. Another approach for improving the results of the testing phase is to automatically generate test cases. Writing test cases manually is an expensive and

tedious work, the automatic generation of test cases can speed up the process and improve the final coverage. In the context of automatic test generation, the model-based testing (MBT) approach has been proposed. The increasing importance of model-based and test-centered development methodologies have brought attention to MBT both in the academic field and in industry.

The use of models in software engineering is constantly increasing due to the possibility of leveraging models to provide an abstract representation of the system, simplifying complex structures and behaviors. Models allow different stakeholders to analyze different aspects of the system from different points of view. Thus, models facilitate decision-making and communication. In model-based testing, the goal is to obtain concrete test cases for the system under test leveraging models that provide an abstraction of the SUT itself. In MBT, a model encodes the intended behaviors of the SUT. Test cases are generated from the model, concretized and then executed on the SUT. The model of the SUT is not only used for test generation, but can also serve as requirements description and specification document. There are several approaches proposed for MBT, often associated with ad-hoc tools. Utting *et al.* [33] describes a generic process of MBT for functional testing, which is currently the main industrial usage of MBT. The process consists of five steps:

Step 1. Build a model (usually called test model) of the SUT from informal requirements or existing specification documents. It is important to have independence between the test model and any development models in order to avoid errors propagating to the generated tests. The test model is validated against the requirements. The test model must provide the correct level of abstraction: it must be precise enough to generate meaningful test cases but not too precise, otherwise, the effort of validating the model would be equal to the effort of validating the SUT.

Step 2. Choose test selection criteria to guide the automatic test generation. Test policies and test objectives are defined and formalized into test plan documents.

Step 3. Transform test selection criteria into test case specifications, which formal-

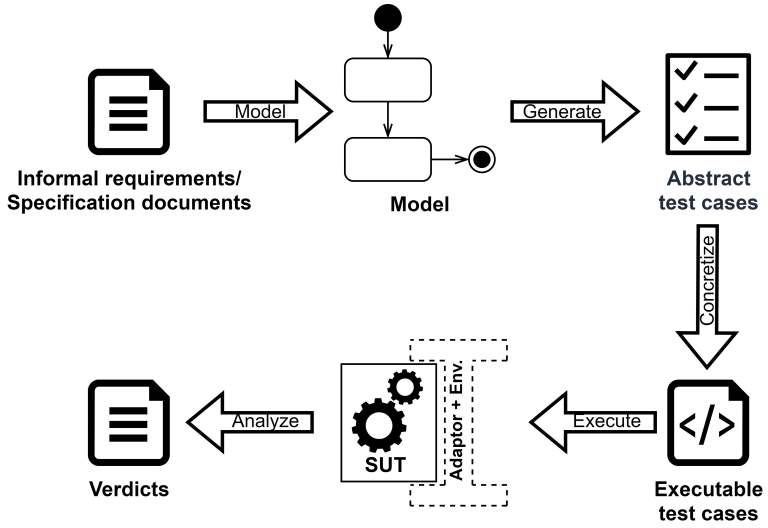

Figure 1.1: A generalization of model-based testing processes.

ize the notion of test selection criteria and make them operational. A test case specification is a high-level description of a desired test case.

Step 4. Generate a set of test cases that satisfy all the test case specifications. Usually, test generators try to minimize the resulting test suite so that a small number of test cases is enough to cover a large number of test specifications.

Step 5. Run the test cases. This phase can be manual or automated by a test execution environment. For the execution of a test case it is necessary to concretize it so that it is runnable on the SUT. The output of the SUT is abstracted to be compared against the expected results (i.e. the model behavior). The component that concretizes the test inputs and abstracts the test outputs is the adaptor.

Li *et al.* [21] describe a similar five-step approach where steps two, three and four are joined in a single "generation of test cases" step. Step five is then divided in three steps: "concretization of test cases", "execution of test cases" and finally "results analysis". This general model-based testing process is shown in Figure 1.1.

This thesis aims to present CREATest and the approach it adopts for the generation of abstract test cases. CREATest is a tool that allows the use of the well-known tool-kit itemis CREATE, formerly known as YAKINDU Statechart Tools, for model-based testing. More specifically, CREATest generates a test class (in the SCTUnit

language) for a given CREATE statechart. Therefore, it accomplishes step four of the process described in [33] or part of step two of the process described in [21]. In addition, it is possible to use the abstract test cases generated by CREATest to validate the model before the execution of the actual MBT process. In MBT it is fundamental for the model to capture the intended behavior of the SUT, and the generated test cases can be used as scenarios to ensure that this property is fulfilled.

Although the main use of a test cases generator for a modeling tool-kit such as itemis CREATE is as part of the model-based testing process, it is also true that the main use of itemis CREATE is for model-driven development (MDD), or model-driven software development (MDSD). The main idea of MDD is to construct the models that describe the system before developing the system itself. After that, the models are used for the generation of source code. It can be assumed that the generated code will be correct as long as the model is correct. To improve the confidence that the model is correct, it is possible to unit test it. Therefore, an automatic test generator such as CREATest could also be used in model-driven development, especially when the software needs to be accompanied by test cases (e.g. for certification or regression testing) as it could speed up testing of the models and improve coverage.

In summary, the purpose of CREATest is to provide a test generator for itemis CREATE, a feature that is missing from the tool-kit as well as in academic research.

The main goal of this work is to suggest a new paradigm for the generation of abstract test cases and to provide a practical scenario where this approach is implemented, specifically in a tool for generating SCTUnit test classes for CREATE statecharts. The idea is to translate the state machine to code and exploit a tool for source code test generation. Then, abstract test cases at the same abstraction level of the input model are derived from the generated test cases. In particular, in CREATest, the selected programming language is Java. The itemis CREATE code generator is used for translating the model to source code while EvoSuite [16] is used for the generation of test cases.

The thesis is structured as follows: Chapter 2 introduces a small overview of state machine formalisms and presents itemis CREATE and EvoSuite. Chapter 3

presents some model-based testing state-of-the-art tools and the novelties introduced in this work. Chapter 4 introduces the basic process and the main ideas behind CREATest. Chapter 5 discusses the implementation of CREATest. In Chapter 6, the experiments and their results are presented. Chapter 7 presents the limitations of the tool and directions for future work. Chapter 8 provides a brief conclusion. Finally, Appendix A is a brief user guide to CREATest.

Chapter 2

Background and Tools

This chapter aims to provide some background information so that the discussion that is presented later can be easily understood. At the moment, finite-state machines and all their variations are probably the most used formalism to represent models in MBT and MDD. itemis CREATE, and hence CREATest, uses a variation of finite-state machines as a modeling formalism. For this reasons, an overview of the most used state machine formalisms is provided here. In addition, to understand where CREATest fits, it is also necessary to provide a brief overview of itemis CREATE and its features. Finally, an analysis of EvoSuite, the core tool responsible for the generation of test cases in CREATest, is presented.

2.1 State Machines formalisms

A state machine is a behavioral model that consists of a finite number of states [36]. For this reason, state machines are also called finite-state machines (FSM). State machines are based on the concepts of states and transitions: given the current state and an input, the machine performs a transition to another (or even the same) state. Depending on the state machine type, outputs can be produced. Only deterministic state-machines are considered in this overview. Formally [15], a finite-state machine is a 5-tuple $(\Sigma, S, s_0, \delta, F)$, where:

- Σ is the input alphabet, a finite non-empty set of symbols;

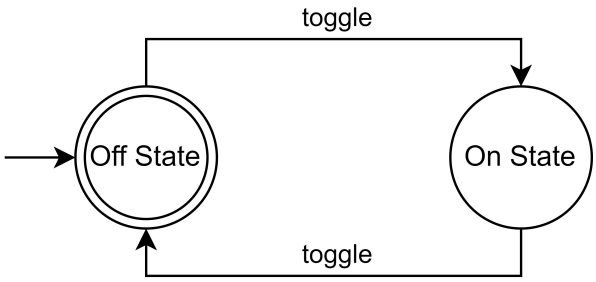

Figure 2.1: A simple finite-state machine.

- S is the finite non-empty set of states;
- s_0 is the initial state, $s_0 \in S$;
- δ is the state-transition function, $\delta : S \times \Sigma \to S$;
- F is the set of final states (possibly empty), $F \subseteq S$.

Figure 2.1 shows a simple finite-state machine example. Usually, an arrow with no starting state is used to graphically represent the initial state and final states are represented with double circles. Thus, in Figure 2.1, "Off State" is both the initial state and the only final state. The mathematical representation of the FSM in figure is:

- $\Sigma = \{\text{toggle}\}$;
- $S = \{\text{Off State, On State}\}$;
- $s_0 = \text{Off State}$;
- $\delta(\text{Off State, toggle}) = \text{On State}$,
  $\delta(\text{On State, toggle}) = \text{Off State}$;
- $F = \{\text{Off State}\}$.

The concept of state machines can be dated back to Turing machines, proposed by Alan Turing in 1936 [3]. A Turing machine is composed of a tape, a head, a table, and a state registry. Turing machine formalization can be considered a type of state machine since, at any point in time, a Turing machine is at one of a finite number of

states. Therefore, any Turing machine can be modeled using modern state machine diagrams. After Turing machines, different formalisms have been proposed, starting from the basic types known as Moore machines and Mealy machines, both proposed in 1956. After that, Harel statecharts have been proposed in 1987. At present, the scenario is dominated by UML state machines, originated in the early 1990s. All these formalisms, along with EFSMs and ASMs, used in some of the tools presented in Chapter 3, are presented here.

2.1.1 Moore machines

Moore machines extend the basic concept of finite-state machines adding outputs to states. The output does not depend on any input and is only determined by the current state. In addition, Moore machines have no final states. Formally [22], a Moore machine is a 6-tuple $(\Sigma, O, S, s_0, \delta, \lambda)$, where:

- Σ is the input alphabet;
- O is the output alphabet;
- S is the finite non-empty set of states;
- s_0 is the initial state, $s_0 \in S$;
- δ is the state-transition function, $\delta : S \times \Sigma \to S$;
- λ is the output function, $\lambda : S \to O$.

The left side of Figure 2.2 shows a simple Moore machine example. The convention for the states is "state_name / output". In the example Σ and O coincide.

2.1.2 Mealy machines

Similarly to Moore machines, Mealy machines are also able to produce outputs. In a Mealy machine, transitions can produce outputs, while states cannot. As Moore machines, Mealy machines have no final states. Formally [22], a Mealy machine is a 6-tuple $(\Sigma, O, S, s_0, \delta, \lambda)$, where:

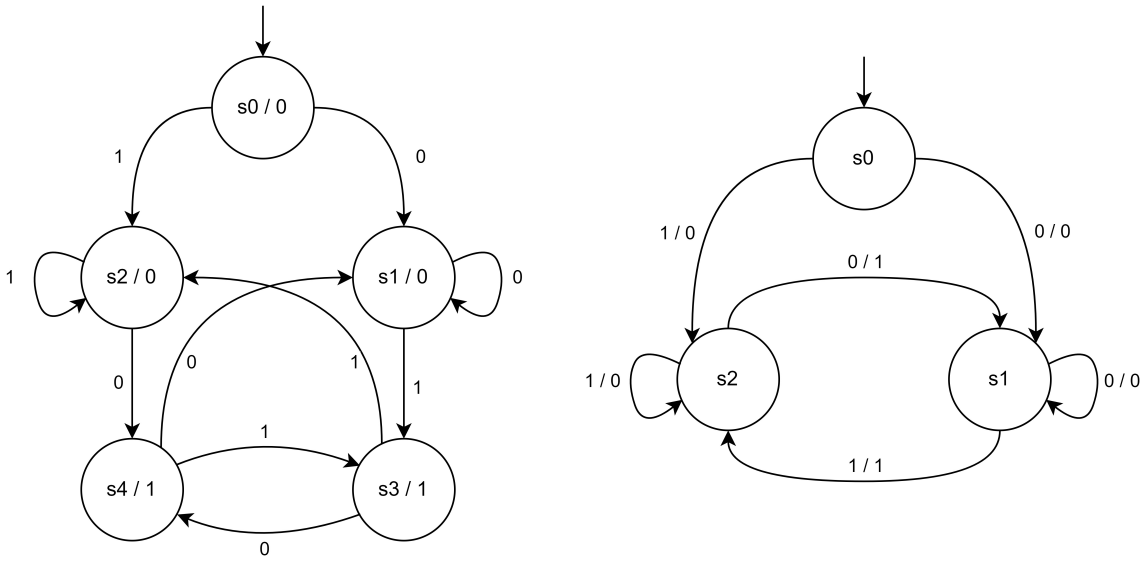

Figure 2.2: A simple Moore machine (on the left) and the equivalent Mealy machine (on the right).

- Σ is the input alphabet;
- O is the output alphabet;
- S is the finite non-empty set of states;
- s_0 is the initial state, $s_0 \in S$;
- δ is the state-transition function, $\delta : S \times \Sigma \to S$;
- λ is the output function, $\lambda : S \times \Sigma \to O$.

Every Moore machine can be translated into an equivalent Mealy machine and vice versa. Typically, a Mealy machine needs less states compared to the equivalent Moore machine, as shown in Figure 2.2. The right-hand side of Figure 2.2 shows a simple Mealy machine alongside its equivalent Moore machine (on the left). The convention for transitions is "input / output".

2.1.3 Extended finite-state machines

Intuitively, extended finite-state machines (EFSM) augment finite-state machines with the notions of memory and variables. In EFSMs, transitions between states

are also associated with a guard. Guards represent a condition that must hold with respect to the variables in the memory [35]. Like Mealy machines, output is produced on transitions. Different mathematical representations for EFSM have been proposed over the years. The formalization proposed in [6] is relatively simple; an extended finite-state machine is a 7-tuple $(\Sigma, O, S, D, F, U, T)$, where:

- Σ is the input alphabet;
- O is the output alphabet;
- S is the finite non-empty set of states;
- D is an n-dimensional space $D_1 \times D_2 \times ... \times D_n$;
- F is the set of enabling functions f_i, such that $f_i : D \to \{0, 1\}$;
- U is the set of update transformations u_i, such that $u_i : D \to D$;
- T is the transition relation, $T : S \times F \times \Sigma \to S \times U \times O$.

Intuitively, D represents the memory (i.e. the variables), F the guards and U the update of the memory. The transition function T models transitions: given the state, the input symbol and the boolean evaluation of the guard, a transition is taken (i.e. a new state is reached, the variables are updated and an output is produced). Other mathematical representations, like the one used to define EFSMs in [35], includes:

- s_0 is the initial state, $s_0 \in S$;
- F is the set of final states (possibly empty), $F \subseteq S$.

2.1.4 Harel statecharts

Also using Mealy machines (or EFSMs), the number of states required for modeling complex systems tends to explode. Harel [18] coined the term "statechart" and proposed a modular, hierarchical and well-structured approach for modeling complex systems. The approach has been initially described in its diagrammatic terms and no algebraic description has been provided. The main introduced concepts are depth

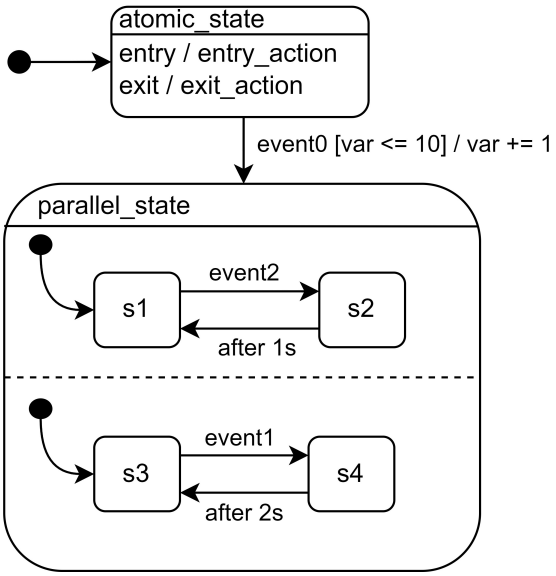

Figure 2.3: A simple Harel statechart.

and orthogonality, in fact, in [18] statecharts are defined as follows:

statecharts = state-diagrams + depth + orthogonality + broadcast-communication

Starting from *depth*, states can be organized in hierarchies, in fact, a state can be defined inside another. A state with sub-states is called composite state or compound state. Focusing on *orthogonality*, composite states can be divided into regions. The states of this type are called parallel states, and the regions are called orthogonal regions. Each region defines its own state machine. When a parallel state is entered, all the state machines of the orthogonal regions are entered and executed in parallel. Moore and Mealy machines output is substituted with the more powerful concept of action. In Harel statecharts, both states and transitions can perform actions. For states, it is also possible to define if the action is performed when entering or when leaving the state. In addition, transitions can be guarded (similarly to EFSMs). A transition is defined as "event [guard] / action". A transition is evaluated when the event occurs. When evaluated, if the guard evaluates to true the transition is executed the action is performed. Thanks to guards, more than one transition can be evaluated when an event occurs, but no more than one will be executed. This behavior, in addition to automatic transitions and delayed transitions, represents the *broadcast-communication* addend in the formula. In addition to these features, Harel statecharts introduce variables (and so a memory, similarly to EFSMs), history

states, temporal logic, as well as entry and exit points, in order to facilitate the modeling of complex systems. Figure 2.3 shows a simple Harel statechart made of an atomic state (a state containing no sub-states) and a parallel state containing two orthogonal regions.

2.1.5 UML state machines

The Unified Modeling Language (UML) is a general-purpose, semi-formal, graphical modeling language based on the object-oriented approach. The latest version available at the moment is UML 2.5. The language defines two major kinds of UML diagrams: structure diagrams and behavior diagrams. As the names suggest, structure diagrams model the static structure of the system and its components on different levels of abstraction, whereas, behavior diagrams show the dynamic behavior of parts of the system. Some of the most used structure diagrams are class diagrams, component diagrams and deployment diagrams. For behavior diagrams, the most used are use-case diagrams, sequence diagrams (that are also a kind of interaction diagrams) and activity diagrams.

UML provides a standard notation also for state machines. UML state machine diagrams, formerly known as UML statecharts, are a kind of behavior diagram. UML state machines are based on Harel statecharts and extend and refine them with object-oriented principles. One of the main object-oriented additions introduced in UML is the possibility to use object procedures instead of actions. For example, it is possible to directly execute Java method calls when a transition fires or when entering a state. This feature allows the state machine or its source code translation to directly interact with other source code. The extension of the already powerful Harel statecharts with object-oriented principles made UML state machines very popular in the software engineering community.

2.1.6 Abstract state machines

Abstract state machines (ASMs) extend the notion of finite-state machines in order to operate over arbitrary data structures. It is possible to give a mathematical

description of ASMs, but given its complexity it falls outside the scope of this thesis and only a quick overview that aims to provide some intuitions is presented. For a user who is not familiar with this kind of formalism, it is possible to see the definition of an ASM as "pseudo-code over abstract data".

An ASM is like a FSM with the addition of generalized states. In fact, generalized states (or abstract states) are more complex than traditional states: in ASMs, a state represents the instantaneous configuration of the system and its environment. In ASMs, transitions apply a set of rules to the previous state (update). For this reason, a transition describes the change of state. ASMs introduce the concept of monitored function and controlled function: monitored functions represent the input of the ASM and controlled functions represent the state and the output of the ASM. Like Harel statecharts or UML state machines, also abstract state machines implement the concept of sub-machines.

2.2 itemis CREATE

itemis CREATE [19], formerly known as YAKINDU Statechart Tools, is a commercial modular toolkit for developing, simulating, and generating executable finite-state machines. It is based on the open-source development platform Eclipse. YAKINDU Statechart Tools was originally an open-source tool, with the original source code available in an archived repository on GitHub [37]. itemis CREATE uses its own kind of state machines, known as CREATE statecharts, which are based on Harel statecharts and are very similar to UML state machines. In the context of itemis CREATE, the term statechart denotes the graphical representation of a state machine but often the two terms are used interchangeably. There exists two execution schemes for CREATE statecharts: event-driven and cycle-based execution. In the cycle-based execution scheme, events are collected and then processed in a run-to-completion step (typically executed periodically). In the event-driven execution scheme, a run-to-completion step is executed each time an event is raised. State machines are contained in statechart model files. The filename extension of these files is .ysc (formerly .sct). The internal format of a state machine file is

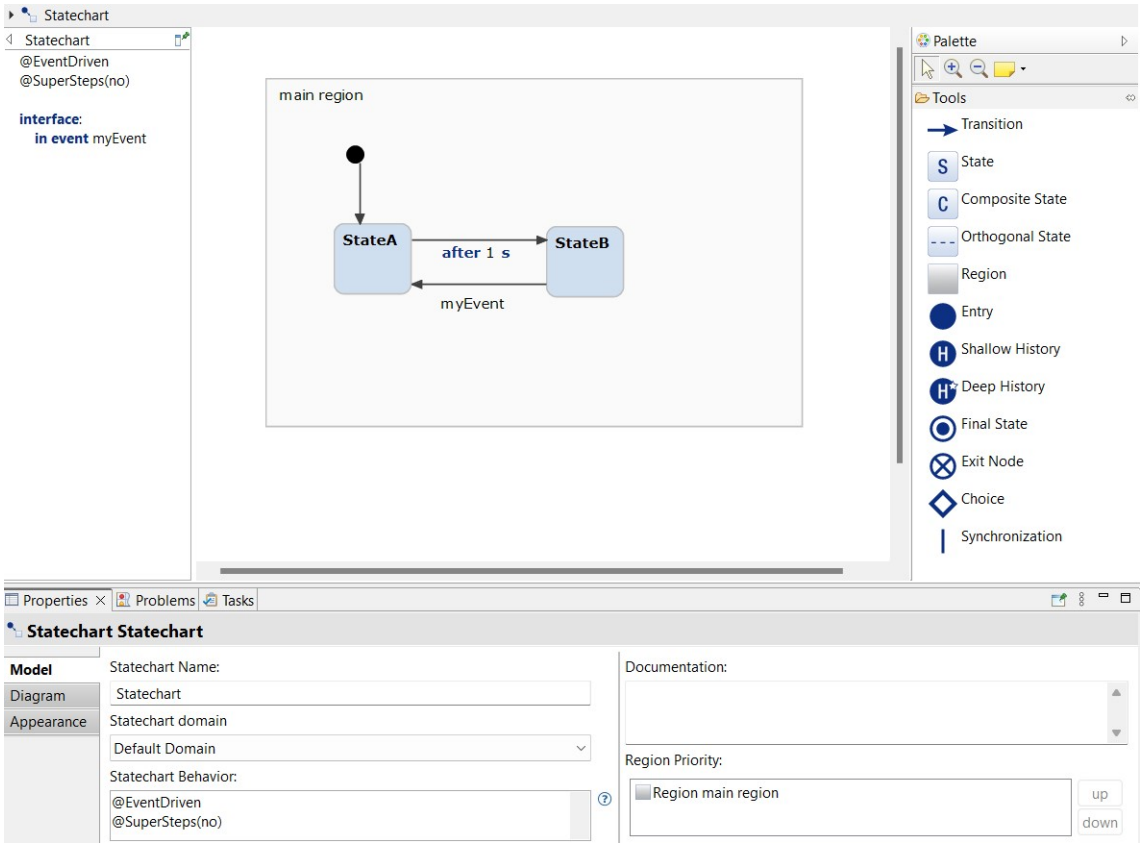

Figure 2.4: A simple state machine opened in the itemis CREATE Eclipse-based editor.

XML Metadata Interchange (XMI). XMI is a framework for defining, interchanging, manipulating and integrating XML data and objects.

The main features that the standard version of itemis CREATE provides are a statechart diagram editor that allows users to graphically create and edit statecharts, a statechart simulator to simulate the behavior of statecharts, a testing framework to test the state machines with unit tests, and code generators for Java, C, C++ and Python. With a professional license, it is also possible to exploit the deep integration with the C/C++ programming languages in order to directly access C/C++ variables, types and operations directly in the statechart. The professional license also provides advanced simulation and debugging features.

itemis CREATE is available as an Eclipse plugin or as a standalone Eclipse-based application. Figure 2.4 shows a basic statechart opened in the editor. Also, a beta cloud editor is available, accessible also as a Visual Studio Code extension. The

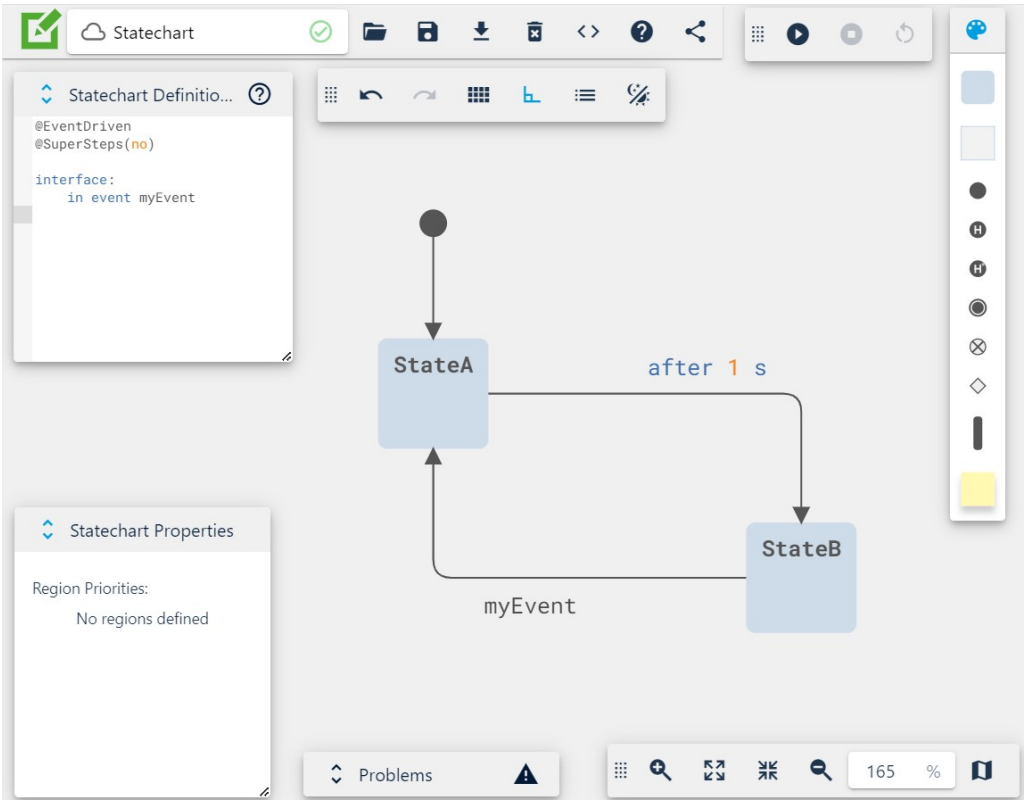

Figure 2.5: A simple state machine opened in the itemis CREATE cloud editor.

cloud editor is easier to be approached by users as it does not require any know-how about Eclipse and it is usable out-of-the-box. Also, the editor itself is more modern and appealing, as shown in Figure 2.5. The format used to define state machines in the cloud editor is JSON and the extension of the file is .scm. At the moment, being still in beta, the cloud editor does not provide all features available in the Eclipse based distribution (whether standalone or plugin). For instance, it appears that the code generator provided in the cloud environment, although showing a really intuitive interface, is not able to generate the source code. In addition, at the moment, no feature for unit testing is available in the cloud editor. CREATest, the tool presented in this thesis, works only if integrated with an Eclipse-based itemis CREATE installation because it exploits the headless code generation, not available in the cloud environment. For this reason, along with the absence of the testing framework, CREATest cannot be integrated with the itemis CREATE cloud editor at the moment. It is not even possible to edit a statechart in the cloud editor and move it to the Eclipse-based environment in order to exploit CREATest given the

different formats used to represent the statecharts in the two environments.

From now on, only the Eclipse-based itemis CREATE release will be considered. First, a brief overview of all itemis CREATE features will be discussed. After that, the focus will be on the two most important features with regard to CREATest: the Java code generator and the SCTUnit testing framework. If one wants to go into details, comprehensive documentation of itemis CREATE, which includes tutorials, videos, examples, and a very informative user guide, is available at [19].

2.2.1 Features overview

This is a brief overview of the features that itemis CREATE offers:

- **Creating and editing statecharts**. The statechart diagram editor allows users to create and edit statecharts by means of a practical GUI that allows drag-and-drop of the basic elements of a statechart, such as regions, states, pseudo-states and transitions. The editor also allows to edit the definition section, a textual area where the user must define the entities that are used in the statechart and the execution scheme (event-driven[1] or cycle-based[2]). In Figure 2.4, the definition section is located on the left of the canvas. It is also possible to edit the appearance of the elements, changing, for example, the font of texts, the color of states and the shape of transitions. The editor also has an integrated validator to check for syntactical and semantical problems in the statechart model: errors and warnings are flagged where they originate, whether in the definition section or in the statechart. The editor also comes with a set of other minor functionalities such as refactoring, statechart comparison and editing proposals.

- **Simulating statecharts**. itemis CREATE supports model simulation. The simulation is a powerful tool that allows users to better understand the behavior of a statechart. The user needs to manually raise standard events (not time based) while the simulation engine takes care of triggering time-based events.

[1] A run-to-completion step is executed each time an event is raised.

[2] Events are collected and then processed in a run-to-completion step.

During the simulation, the active states are highlighted and all the current values of the variables are registered. It is possible to run multiple state machines in parallel or multiple instances of the same state machine. With a professional license it is also possible to run the simulation in debug mode and insert breakpoints into elements so that the simulation automatically pauses when the element in the statechart is reached.

- **Generating state machine code**. itemis CREATE provides a set of code generators for a set of widely used programming languages: C, C++, Java and Python. Regardless of the target programming language, the generated code is a one-to-one mapping of the statechart. The concretization of a statechart into source code allows sending and receiving events to and from the state machine, collecting information about the current active states, setting and getting state machine variables, and making the state machine invoke specific behaviors that are external to it. It is the task of the client code to handle timers, concurrency, scheduling and when events are delivered to the state machine. In order to generate the source code of one or more statecharts it is necessary to create an SGen model. An example of an SGen model is shown in Listing 2.1. An SGen model allows the user to define the target programming language and configure the code generator. Depending on the target language, the available features may change. Some features are common to all generators, they are: *Outlet* feature (with two required parameters: *targetProject* and *targetFolder*, and two optional parameters: *libraryTargetFolder* and *skipLibraryFiles*), *OutEventAPI* feature, *LicenseHeader* feature, *FunctionInlining* feature, that enables the inlining of expressions instead of generating separate functions or methods, and *Debug* feature. In order to read a detailed description of these features, consult the user user guide at [19]. The SGen model must be contained in a file with .sgen extension. Once the SGen model has been defined, it is possible to generate the source code along with all the needed dependencies. itemis CREATE additionally provides a headless code generator infrastructure that allows users to generate source code without interacting with the user interface. The headless code generator can be called

from a command line and thus can be easily integrated with all continuous integration tools. As for traditional code generation, an SGen model specified in a file with .sgen extension must be defined. A deeper analysis of the Java code generator and the structure of the generated Java classes is presented in subsection 2.2.2.

- **Generating SCXML documents and images**. It is also possible to define an SGen model for the generation of images and State Chart XML (SCXML) documents. SCXML is an XML based mark-up language that provides a generic state-machine-based execution environment based on Harel statecharts. SCXML generation is available only with a professional license. Regarding image generation, the image formats available are BMP, PNG, JPG, JPEG, SVG and PDF.

- **C/C++ code integration**. itemis CREATE professional license comes with the *deep C/C++ integration feature*, which allows the use of C/C++ types, variables, and operations directly within the statechart model. With this feature it is possible to directly import in the definition section C/C++ header files in order to directly access all contained type and operation declarations.

- **Modeling multi state machines**. itemis CREATE allows to model a system as the composition of multiple collaborating state machines. Modeling a complex system using different small state machines helps with separation of concerns, state machines reuse and maintainability. The collaboration of state machines is obtained by allowing a state machine to import another. After importing a state machine (called sub machine), it is possible to define variables with the type of the imported state machine. At this point it is possible for a state machine to raise in events on the referenced state machines, react to out events from referenced state machines, get and set variables of the referenced state machines.

- **Testing state machine**. See subsection 2.2.3.

- **Producing execution traces**. itemis CREATE provides the YAKINDU

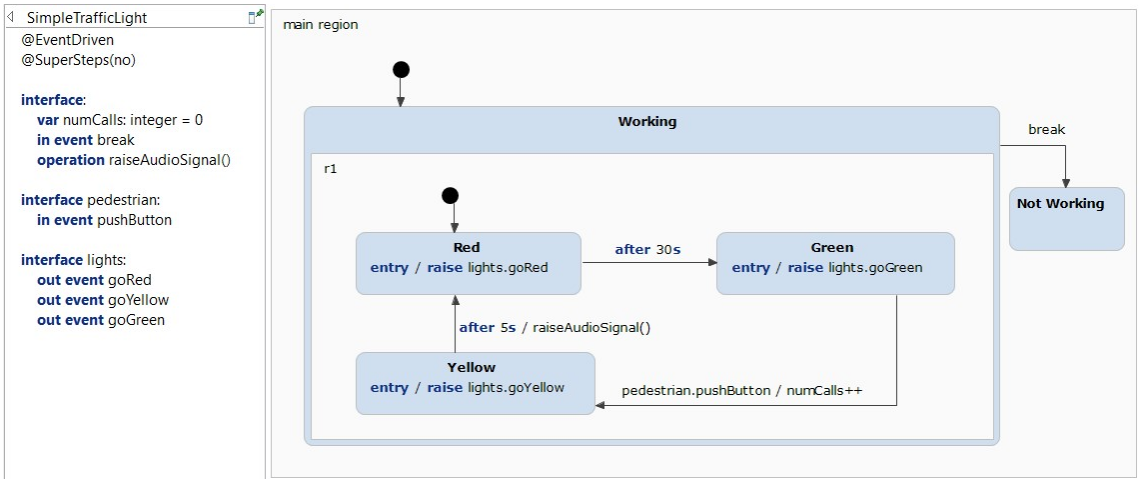

Figure 2.6: Statechart example model: simple traffic light.

Execution Tracing (YET), an infrastructure that enables information about the execution of remote state machines to be provided during the execution itself. YET provides open formats, protocols, and APIs. It can be used for debugging state machines, execution analysis, testing and co-simulation.

2.2.2 The Java code generator

The Java code overview is accompanied by an example. The statechart used is shown in Figure 2.6 and models a simple traffic light. Besides being simple and not very meaningful, the statechart defines some interesting insights to understand the general structure of the generated Java class. For example, the statechart defines two named interfaces, a variable, timed event triggers, in and out events and an operation.

The generation of Java code from the statechart starts with the definition of the SGen model. It is necessary to specify at least the *Outlet* feature with the parameters *targetProject* and *targetFolder*. In addition to the optional features common to all generators, Java code generators can specify additional optional features: *Naming* feature (the possible parameters are: *basePackage*, *libraryPackage* and *typeName*), *GeneralFeatures* feature (all boolean parameters: *RuntimeService* and *TimeService* allow the generation of additional code, *synchronized* adds the synchronized keyword where it is appropriate, and *runnable* makes the state machine implement

java.lang.Runnable), and *Tracing* feature (boolean parameters that enable the generation of tracing callbacks). For the example described here, it is sufficient to define the SGen model as follows:

```
GeneratorModel for create::java {
        const PROJECT : string = "TargetStatecharts"
        const FOLDER : string = "src"
        statechart SimpleTrafficLight {
                feature Outlet {
                        targetProject = PROJECT
                        targetFolder = FOLDER
                        libraryTargetFolder = FOLDER
                }
                feature Naming {
                        basePackage = "implementation"
                }
                feature GeneralFeatures {
                        TimerService = true
                }
        }
}
```

Listing 2.1: SGen Model example.

This example shows also how to use constants in the definition of the SGen model. The *Outlet* feature specifies that the project where to write the generated artifacts is *TargetStatecharts* and the folder, within the target project, where to write both model-dependent code (*targetFolder*) and model-independent code (*libraryTargetFolder*) is *src*. The *Naming* feature specifies that the package name for the generated statechart class is *implementation*. Finally, the parameter *TimerService* set to true in *GeneralFeatures* enables the generation of a timer service implementation. Since no *libraryPackage* is defined, the library files (model-independent code) package is *com.yakindu.core*.

The following list is a quick overview of all the possible library files that can be generated by the Java code generator, and is intended to give an intuition of how to use them. A more detailed description can be found in the user manual available at [19].

- **IStatemachine.java**: contains the interface `IStatemachine`. Each state machine implements either the `IEventDriven` interface or the `ICycleBased` interface. Both of these interfaces extend `IStatemachine`, so every state machine implements `IStatemachine`. It is defined as follows:

```
public interface IStatemachine {
    public void enter();
    public void exit();
    public boolean isActive();
    public boolean isFinal();
}
```

- **IEventDriven.java**: contains the interface `IEventDriven` and it is only generated if the statechart is event-drive. It is only implemented by statecharts that implement the event-driven execution scheme. It is defined as follows:

```
public interface IEventDriven extends IStatemachine{
    public void triggerWithoutEvent();
}
```

- **ICycleBased.java**: contains the interface `ICycleBased` and it is only generated if the statechart is cycle-based. It is only implemented by statecharts that implement the cycle-based execution scheme. It is defined as follows:

```
public interface ICycleBased extends IStatemachine{
    public void runCycle();
}
```

- **ITimed.java**: contains the interface `ITimed` and it is only generated if the statechart uses timed event triggers. It is only implemented by statecharts that deal with timed events. It is defined as follows:

```
public interface ITimed {
    public void raiseTimeEvent(int eventID);
    public void setTimerService(ITimerService timerService);
}
```

- **ITimerService.java**: contains the interface `ITimerService` and it is only generated if the statechart uses timed event triggers. It is implemented by classes whose goal is to provide a timer service for the statechart. It is defined as follows:

```
public interface ITimerService {
    public void setTimer(ITimed callback, int eventID, long time,
        boolean isPeriodic);
    public void unsetTimer(ITimed callback, int eventID);
}
```

- **TimerService.java**: contains the class `TimerService`, a default timer service implementation. It is only generated if explicitly specified in the SGen model. In most cases, it is a sufficient implementation of `ITimerService`.

- **VirtualTimer.java**: contains the class `VirtualTimer`, an alternative timer service implementation. It is only generated if the statechart uses timed event triggers.

- **ITracingListener.java**: contains the interface `ITracingListener` and it is only generated if explicitly specified in the SGen model. Is the task of the client code to provide an implementation of this interface. It is defined as follows:

```
public interface ITracingListener<T> {
    void onStateEntered(T state);
    void onStateExited(T state);
}
```

- **Observable.java**: contains the class Observable, used for the observer mechanism to react on outgoing events. It is only generated if the statechart uses outgoing events.
- **Observer.java**: contains the class Observer, used for the observer mechanism to react on outgoing events. It is only generated if the statechart uses outgoing events.

In the proposed example, the only library file that has not been generated is ITracingListener.java. The following description is intended to illustrate the general structure of a state machine class and how statechart elements are mapped to Java members. In order to achieve this, some code snippets from the example state machine class are used. The state machine implementation class is contained in the SimpleTrafficLight.java file. A state machine class implements either the `IEventDriven` interface or the `ICycleBased` interface and potentially the `ITimed` interface. The `SimpleTrafficLight` class implements `IEventDriven` and `ITimed`:

```
public class SimpleTrafficLight implements ITimed, IEventDriven {
    ...
}
```

Each named interface is translated into an inner class. Elements defined in the unnamed interface are defined directly in the state machine class:

```
public class SimpleTrafficLight implements ITimed, EventDriven {
    public static class Pedestrian { ... }
    public static class Lights { ... }
    protected Pedestrian pedestrian;
    protected Lights lights;
    ...
    public SimpleTrafficLight() {
        pedestrian = new Pedestrian(this);
        lights = new Lights();
        ...
    }
```

```
    ...
    public Pedestrian pedestrian() {
        return pedestrian;
    }
    public Lights lights() {
        return lights;
    }
    ...
}
```

Only public fields and methods should be accessed by the client code. Statechart variables are translated into fields with associated getters and setters:

```
public class SimpleTrafficLight implements ITimed, IEventDriven {
    public static class Pedestrian { ... }
    public static class Lights { ... }
    ...
    public SimpleTrafficLight() {
        ...
        setNumCalls(0l);
        ...
    }
    ...
    private long numCalls;
    public long getNumCalls() {
        return numCalls;
    }
    public void setNumCalls(long value) {
        this.numCalls = value;
    }
    ...
}
```

Incoming events are translated into methods and outgoing events are translated into observable objects that the client code can subscribe to:

```
public class SimpleTrafficLight implements ITimed, IEventDriven {
    public static class Pedestrian {
        ...
        private boolean pushButton;
        public void raisePushButton() { ... }
        ...
    }
    public static class Lights {
        ...
        private boolean goRed;
        protected void raiseGoRed() {
            goRed = true;
            goRedObservable.next(null);
        }
        private Observable<Void> goRedObservable = new
            Observable<Void>();
        public Observable<Void> getGoRed() {
            return goRedObservable;
        }
        ...
    }
    ...
}
```

Operations are translated into inner interfaces. The client code must provide an implementation of these interfaces and pass an instance of it to the state machine via the provided method:

```
public class SimpleTrafficLight implements ITimed, IEventDriven {
    public static class Pedestrian { ... }
```

```
    public static class Lights { ... }
    ...
    public interface OperationCallback {
        public void raiseAudioSignal();
    }
    private OperationCallback operationCallback;
    public void setOperationCallback(OperationCallback operationCallback) {
        this.operationCallback = operationCallback;
    }
    ...
}
```

The states are translated into an enumeration within the state machine class. Each enum constant represents one state. The constant is the sequence of regions and states (separated by an underscore) that constitute the hierarchy of the state itself. In addition, an enum constant representing the null state (`$NULLSTATE$`) is added:

```
public class SimpleTrafficLight implements ITimed, IEventDriven {
    public static class Pedestrian { ... }
    public static class Lights { ... }
    ...
    public enum State {
        MAIN_REGION_WORKING,
        MAIN_REGION_WORKING_R1_YELLOW,
        MAIN_REGION_WORKING_R1_GREEN,
        MAIN_REGION_WORKING_R1_RED,
        MAIN_REGION_NOT_WORKING,
        $NULLSTATE$
    };
    ...
}
```

The members of the Java class with public visibility make the Java class a one-

Table 2.1: Mapping between itemis CREATE types and Java types.

CREATE type	**Java type**
integer	long
real	double
boolean	boolean
string	String
void	void

to-one mapping of the statechart. The state machine class is completed by a set of protected and private fields and methods whose job is to actually implement the behavior of the state machine. The Table 2.1 shows how the default types in CREATE statecharts are mapped to Java types. It is possible to use the itemis CREATE Java code generator without the user having to call it directly in the Eclipse environment, thus allowing headless code generation. The itemis CREATE installation comes with the statechart compiler, which is a scc.bat file for Windows or a scc file for Linux and MacOS. This is a script file that allows the generation of Java code from the command line.

2.2.3 The SCTUnit testing framework

It is possible to assume that the source code generated from a state machine is correct as long as the state machine itself is correct. For testing state machines, itemis CREATE provides the SCTUnit scripting and testing framework. As the user guide [19] suggests, the main use of SCTUnit is test-driven development. The SCTUnit framework allows the users to create and edit test cases in the SCTUnit language and to execute them against the models under test.

The test cases (or operations) must be defined within a test class. Test classes contain one or more test cases. A test class must be contained in a file with .sctunit extension. A test class controls exactly one state machine. Within a test class, it is possible to define variables and constants that are visible to all the operations of the test class. In addition to these global variables, an operation has access

to: the other operations defined in the test class, its own defined variables (local variables), the elements defined in the state machine controlled by its state class and the entities that this state machine imports. Operations can either return nothing or return an expression. The typical structure of a test case starts with an `enter` statement followed by a sequence of `raise`, `proceed`, and `assert` statements. The `enter` statement initializes and starts the state machine. The `raise` statements are used to raise incoming events. The `proceed` statements are used to make the state machine proceed for a given amount of time. The `assert` statements are used to check whether: a state is active or not (a state must be specified through its complete hierarchy, from the top-level region down to the state itself), an operation has been called (with which parameters and how many times) or not, an outgoing event has been raised or not, the state machine is active or not, and whether the active state is a final state or not. The condition can also be an expression containing variables and constants. At the end, it is possible to exit the state machine with the `exit` statement. If the execution is cycle-based it is possible to call the `proceed` statement to perform a given number of run-to-completion steps. If the execution is event-driven, it is possible to call the `triggerWithoutEvent` statement to perform a run-to-completion step without raising any event. The SCTUnit language also provides conditional statements (`if` and `if else`) and loop statements (`while`). Finally, it is is possible to mock operation calls.

It is possible to aggregate a set of test classes into a single test suite. The execution of a test suite results in the execution of all the test cases within all the test classes specified in the test suite. Test classes and test suites can be organized in different namespaces. The test class shown in Listing 2.2 provides two simple hand-written test cases for the SimpleTrafficLight statechart shown in Figure 2.6.

It is possible to generate SCTUnit tests as source code. The supported programming languages are C, C++, Java, Python as well as SCXML. To generate the source code of an SCTUnit test class, it is necessary to define an SGen model. An SGen model for tests has the same structure as an SGen model for statecharts, with some differences in the available features. The SCTUnit code generators translate the SCTUnit test cases into a unit test framework of the target language. For ex-

```
testclass SimpleTrafficLightTest for statechart SimpleTrafficLight {
    @Test
    operation workingStateTest() {
        enter
        assert active(main_region.Working.r1.Red)
        proceed 35 s
        assert active(main_region.Working.r1.Green)
        raise pedestrian.pushButton
        assert active(main_region.Working.r1.Yellow)
        proceed 10 s
        assert called raiseAudioSignal() 1 times
        assert lights.goRed
        assert !is_final
        exit
    }
    @Test
    operation notWorkingStateTest() {
        enter
        assert active(main_region.Working.r1.Red)
        raise break
        assert active(main_region.Not_Working)
        exit
    }
}
```

Listing 2.2: An example of SCTUnit test class for the statechart shown in Figure 2.6.

ample, JUnit is used as the test framework and Mockito is used to mock methods when the target language is Java.

When an SCTUnit test class (or test suite) is executed, information about the status of each test case (i.e. which test cases failed and why) is provided. In

addition, test coverage metrics are computed and can be examined in the coverage view. Coverage is computed for each element in the statechart and is determined as follows:

- transition coverage is 0% if the transition is never executed, 100% if it is executed at least once;
- state coverage is 0% if the state is never entered, and it can go up to 100% depending on how many outgoing transition have been executed at least once;
- region coverage is the weighted average coverage of all included states;
- statechart coverage is the weighted average coverage of its top level regions.

itemis CREATE provides a coverage highlighting feature that colors the states of the tested statechart depending on their coverage. A state is colored green in case of 100% state coverage, red in case of 0% state coverage, and yellow otherwise. Running a test class (or test suite) produces a coverage file with a .cov extension. The coverage file can be used to compare the current coverage results with past coverage results. It is also possible to export an HTML report of the test coverage. The report includes all coverage information and a highlighted image of each covered statechart.

2.3 EvoSuite

EvoSuite [16] is an open-source tool that automatically generates JUnit test suites for Java classes. EvoSuite implements a white-box, evolutionary, and search-based approach. In the automatic test cases generation for source code there is the oracle issue, i.e. how to verify that the output of test cases is the expected output. If the oracle is missing, that is the case of EvoSuite, faults can only be automatically detected if they lead to program crashes, deadlocks, or violate a formal specification. In all other cases, all generated tests will pass regardless of whether the software is correct or not. To mitigate the oracle issue, EvoSuite tries to generate test suites that are as small as possible. If the generated test suite is small in terms of test data

and assertions, it is possible for the user to manually verify the generated code, i.e. to check whether the assert statements are consistent with the program semantics.

In white-box testing a common approach to test case generation is to select a coverage goal for each given coverage criterion and derive a test case that exercises the goal itself. The main problem with this approach is that it assumes that coverage goals are equally important, equally difficult to achieve, and independent. Since these assumptions do not hold, the choice of coverage goals order affects the overall quality of the generated test cases.

To overcome this limitation, the whole test suite generation approach has been introduced in EvoSuite. In this approach, the whole test suite is evolved with respect to the overall code coverage rather than individual coverage goals. In the latest versions, the many-objective approach, implemented in the DynaMOSA algorithm, has replaced the whole test suite generation approach. DynaMOSA is a many-objective genetics search-algorithm that optimizes multiple coverage criteria at the same time, treating different test coverage requirements as distinct and contrasting search objectives.

There are several tools that allow the automatic test case generation for a given programming language. For Java, Randoop and Evosuite are probably the most well-known tools. Some of these tools compete annually at the International Workshop on Search-Based Software Testing (SBST). EvoSuite has proven its leading role as an automated test generation tool for Java, achieving the highest overall score in nine out of the ten editions in which it has participated. For example, in the 2021 edition [34], six tools (including Randoop) were evaluated on a set of 98 Java classes in terms of code coverage and mutation scores. EvoSuite achieved the highest overall score.

EvoSuite is available as an executable jar file that comes with a large set of options and parameters that allow highly customizable generation of test cases. EvoSuite is also available as a Maven plugin, IntelliJ IDEA plugin, Eclipse plugin and Jenkins plugin (the latter two are still experimental). The following two subsections delve deeper into the two main concepts underlying Evosuite [16]: evolutionary search and mutation testing.

2.3.1 Evolutionary search

Evolutionary search approaches attempt to evolve a population of candidates (or individuals) using operations that mimic the natural mechanisms of evolution, such as selection, crossover and mutation. Individuals are selected based on their fitness (an estimate of how close a candidate is to the optimal solution). Evolutionary operators are applied to these selected individuals (parents) to produce a new generation (offspring) that completely replaces the previous population. Otherwise, it is possible to keep the best candidates from the previous population in the new one (elitism). The initial population is usually obtained randomly. Crossover operators take two or more parent individuals and combine them to obtain an equal number of new individuals. Mutation operators take one individual and randomly modify it in order to obtain a new individual. As the generation passes, the population evolves and the overall fitness increases. The schema shown in Figure 2.7 describes the general structure of an evolutionary algorithm as described in [12].

In EvoSuite, a candidate consists of a test suite made of a variable number of test cases. Crossover consists in randomly exchanging test cases between two test suites. Mutation consists in adding a new test case to the test suite or mutating an existing test case. Test case mutation consists of adding, deleting or changing individual statements or parameters. The fitness of an individual is measured with respect to a coverage criterion. At the end of the search, the final test suites are minimized in order to keep only the statements that contribute to the coverage.

During the evolutionary search, test cases are executed to measure the fitness value. This interaction with the code could be dangerous (for example, it could cause data loss), specially in the initial iterations when the initial test suite is random. For this reason, EvoSuite provides a security manager that prevents unwanted access to the environment.

2.3.2 Mutation testing

The test oracle problem is one of the main problems in traditional white-box test generation. A fault in the tested program can only be detected if the user man-

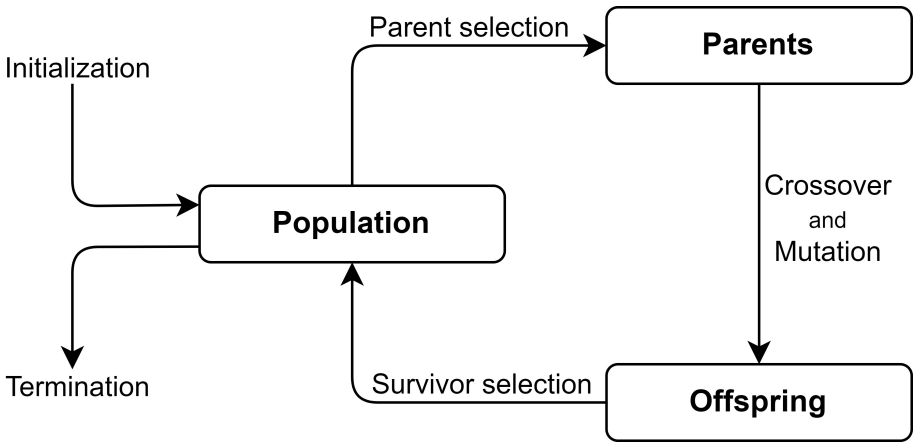

Figure 2.7: The general scheme of an evolutionary algorithm as described in [12].

ually verifies the correctness of the generated test suite and identifies a problem. Otherwise, a test case may fail at a later point (regression failure). For this reason, EvoSuite implements a mutation testing approach to identify the important and effective assertions. In mutation testing, defects (also called mutants) are artificially introduced into a program. The test cases are evaluated on both the original and the mutated programs. A mutant is detected (killed) if an assertion fails on the mutated program and passes on the original program. Mutants that are not detected (survived mutants) denote a problem in the test suite and a new test case should be added or an existing test case should be modified. EvoSuite uses mutation testing to determine which assertions are sufficient to detect mutants, i.e. which assertions are most likely to be effective.

Chapter 3

Related Works

It is known that software testing is a crucial aspect in the development of complex software systems, especially in the safety-critical domain. The automatic generation of test cases can significantly reduce the resources needed for this process and improve the effectiveness of the testing phase. Model-Based Testing (MBT) has proven to be a powerful technique in this field. Consequently, many techniques and tools have been developed in the last few years. This chapter present some of these tools, without any claim to completeness. This brief overview aims to demonstrate the necessity of the tool introduced here and the gaps it addresses. The focus is primarily on the modeling formalism used to capture the SUT, the format of the generated test cases and the effort required by users to learn and use the tool.

3.1 State-of-the-art tools

3.1.1 EvoMBT

EvoMBT [14] is an open-source tool that adopts extended finite-state machines (EFSM) as its modeling formalism. It has been developed considering the lack of practical solutions for the test generation for 3D games and other complex software systems. It is based on the search algorithms provided by the well-known EvoSuite [16] tool. The abstract test generation is independent from the SUT. Once the user models the SUT (or part of it) as an EFSM, EvoMBT is able to generate

the abstract test cases. The user must provide a concretization mechanism for the generation of executable test cases runnable on the SUT. Additionally, EvoMBT provides an LTL-based lightweight model checking tool.

Currently, the user can define the input EFSM only as a Java class by implementing the given interfaces. The produced abstract test cases are presented to the user as a list of instances of a given Java class. Thus, users not familiars with Java will have to spend a non-negligible amount of time in order to be able to use EvoMBT. Even experienced Java developers will need to dedicate a considerable amount of time to implement a model, given the verbosity of Java. For these reasons, a graphical editor should be provided, as the authors state in [14]. A comprehensive user documentation is available at [13].

3.1.2 GraphWalker

GraphWalker [17] is an open-source model-based testing tool. It allows the user to model the system under test as a directed graph where edges represent actions on the SUT and nodes represent verifications (i.e. assertions on the SUT). Tests are paths (lists of pairs of edges and vertices) on the graph. GraphWalker navigates the directed graph through random walks and generates tests until a stopping condition defined by the user is met. It can be used both for online and offline testing. Since the format of the abstract test cases is a list of pairs of element (path over the directed graph) stored in a file, it is the duty of the user to concretize them.

GraphWalker is available in three versions: GraphWalker Studio, GraphWalker CLI, and the Eclipse plugin GW4E. In both GraphWalker Studio and GW4E the model can be implemented using a practical GUI. Figure 3.1 shows an example of a model in GraphWalker Studio. An additional tool for modeling the SUT is needed in order to use the GraphWalker CLI and the model format must be JSON or graphML, the generated test cases are in JSON format. As stated in [38], GW4E supports limited stopping criteria compared to GraphWalker Studio and GraphWalker CLI but it is more user-friendly in terms of debugging, abstract test execution, and generation of useful information for testers and developers.

GraphWalker has been selected as the model-based testing tool for the com-

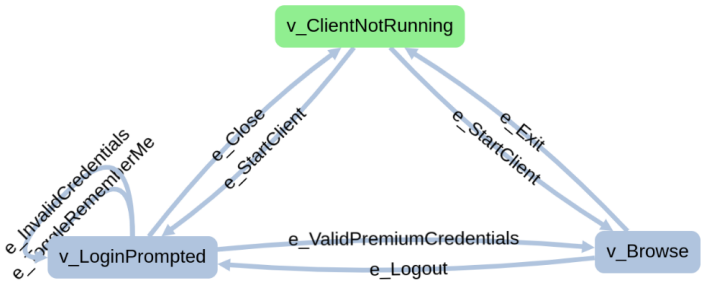

Figure 3.1: An example of GraphWalker model.

parison between MBT and manually written tests in an industrial cyber-physical scenario in [38]. The study shows that the test cases generated by GraphWalker can provide higher coverage than manually written test cases and that GraphWalker can achieve a 100% edge coverage. Comprehensive user documentation with different and meaningful examples is available at [13].

3.1.3 Modbat

Modbat [1] is an open-source model-based testing tool that implements a black-box approach. It is able to model event-driven or input/output-driven systems. In particular, Modbat supports non-blocking I/O operations, operations that throw exception in case of disrupted communication and non-determinism in system actions. Like EvoMBT, Modbat also uses extended finite-state machines as modeling formalism. It also provides a domain-specific language (DSL) embedded in Scala for the definition of the EFSMs. It can be used both for online and offline testing.

Modbat explores the defined model using random search and executes the SUT in tandem. Modbat is able to generate JUnit test cases or it can use a stand-alone format for test cases. As stated in [1], Modbat is similar to earlier MBT tools like ModelJUnit and ScalaCheck, but it provides a significant more concise notation, making it a more user-friendly choice for model-based testing. Modbat also provides visualization for the abstract model and for coverage measurements (state and transition coverage are supported). It is distributed as a JAR file and thus users can interact with it only through a command line interface. Detailed documentation is available at [24].

3.1.4 Spec Explorer

Spec Explorer [31] is a commercial model-based testing tool developed by Microsoft. It extends Microsoft Visual Studio IDE. Spec Explorer uses as modeling formalism Microsoft model programs, based on abstract state machines written in the Abstract State Machine Language (ASML). Model programs can be defined using the C# programming language or any other .NET language. Model programs represent a powerful modeling formalism, being able to implement both structural models and behavioral models, and, for the latter, both the interaction-oriented modeling style and state-oriented modeling style. In particular, Spec Explorer uses state-oriented model programs, in addition to a behavioral descriptions. The behavioral description is a script written in the scripting language Cord that is used to express the behavior of the model, defining configurations and actions. Model programs can be written by hand or derived from existing code.

The generated abstract test cases are in the form of C# files and thus can be consumed by the test framework Visual Studio Test Tools (or any other .NET test framework). SpecExplorer has been the subject of a comparison with Conformiq's Qtronic in [28], and the authors found that Spec Explorer is reliable and easy to use, moreover, the provided model notation is powerful and covers a wide range of features; it is suitable for component, sub-system, and system level testing and it supports regression testing. They also present some critical issues for Spec Explorer: it is not possible to specify the coverage criteria (the only possible strategies are short test and long test), the user must be familiar with .NET and even in that case learning the syntax of Cord scripts will require additional effort, finally, the documentation, available at [31], is not adequate.

3.1.5 Qtronic

Qtronic is a commercial model-based testing tool developed by ConformIQ [8]. At the moment, Qtronic is no more available as a standalone product but it is the engine of two separated products named ConformIQ Designer and ConformIQ Creator. The adopted modeling formalism in Qtronic is UML state machines. It is possible

to define state machines in Qtronic using a textual notation, for which a specific superset of Java programming language called Qtronic Modeling Language (QML) is defined, or graphically by means of the internal editor ConformIQ Modeler. It is also possible to import graphical models from third party UML editors. Qtronic allows translating the generated test cases into executable test scripts in languages such as TTCN-3 and TCL, and in HTML as documentation. Generated test cases can be reviewed and analyzed within an Eclipse user interface.

The test generation is driven by model coverage, and a visual overview of the connection between the generated test cases and the covered requirement is provided. In [28], Qtronic has been found out to be suitable for component, sub-system, and system level testing, like Spec Explorer. In contrast with Spec Explorer, Qtronic does not support regression testing. From the point of view of usability, in [28] it is pointed out that modeling simple systems is straightforward but for advanced features additional training is required. Moreover, the use of QML requires some training. The provided user manual is said to be very informative.

3.1.6 Asmeta

Asmeta [2] is an open-source and complete framework for verification and validation. It is based on Abstract state machines (ASMs) and it is composed of several tools. All tools are distributed as JAR files and some of them are also available as Eclipse plugins. The ASMs must be written in the AsmetaL language. Some of the tools provided in the ASMETA framework are: AsmetaXt (which replaced Asmee), an editor for ASMs, AsmetaLc, a compiler/parser for AsmetaL models, AsmetaS, a simulator, AsmetaVis, for the visualization of AsmetaL models, AsmetaSMV, a model checker based on NuSMv [7], and Asm2C++, a C++ code generator for AsmetaL models.

ASMETA can also be used as a model-based testing tool, as shown in [5]. The study shows that, in the given safety-critical scenario, model-based testing can outperform manual testing. In ASMETA, the adopted modeling formalism for MBT is obviously abstract state machines. The framework provides the ASM Tests Generation Tool (ATGT), that generates abstract test cases for an ASMs written in

AsmetaL. The generated test cases are in the AVALLA language, also used to write scenarios in ASMETA. The abstract test generation phase involves a final optimization phase that improves abstract tests readability and translatability (in concrete test cases), maintaining the semantics unchanged. The user must provide a concretization mechanism in order to obtain concrete test cases runnable on the SUT.

The completeness of ASMETA allows users to have all needed tools for implementing V&V techniques, including model-based testing, in a single framework. The use of ASMETA for MBT entails that the user needs to learn the AsmetaL and AVALLA languages, the latter of which requires little effort. Documentation for the ASMETA framework is available at [2].

3.2 Summary and introduced novelties

Model-based testing is a hot topic in Software Engineering and the number of available tools, both open-source and commercial, is increasing. The task of choosing the most suitable tool might initially seem overwhelming. The brief overview presented here can be a useful starting point, although it considers just a subset of all the existing tools and not all the relevant aspects are analyzed. As mentioned at the beginning of this chapter, the adopted modeling formalism and the usability of the tool play a major role in this analysis, at the expense of other aspects, like the performance. The input and output format of a tool are considered a crucial aspect for usability. These aspects can drive the choice of the tool; for example, in a situation where there is a substantial availability of Java expertise, EvoMBT could be a suitable solution.

At the moment, as far as the writer knows, itemis CREATE [19], formerly known as Yakindu, provides no features for model-based testing and no third parties tools have been developed in the academic sphere. itemis CREATE is a well-known and used tool-set for Model-Driven Development (MDD) that provides a lot of useful features such as simulation, testing and source code generation. The absence of an option for model-based testing is a missed opportunity. An automatic test generator for itemis CREATE state machines, along with the other features natively

provided in itemis CREATE, can be adopted for model-based testing and for enhancing model-driven development. The work presented here tries to fill this gap, providing a practical tool for the generation of test cases for CREATE statecharts, that are based on Harel statecharts and are very close to UML state machines.

itemis CREATE is a commercial tool for model-driven development and it is available as an Eclipse plugin or as a standalone Eclipse-based application. Also a beta cloud editor is available, accessible also as a Visual Studio Code extension. In both cases, it is possible to edit state machines by means of a practical GUI. State machines can be simulated and executable C, C++, Java or Python code can be generated. Testing of state machines is done by means of SCTUnit, a scripting and testing framework for writing unit tests for statechart models. itemis CREATE provides source code generation also for SCTUnit test cases. For a more detailed overview of itemis CREATE, see Section 2.2. Using CREATest, it is possible to automatically obtain SCTUnit test cases for a given state machine.

Aside from providing a starting point for model-based testing using itemis CREATE, CREATest introduces a new approach for abstract test generation, that is, concretizing the model in source code, generating concrete test cases for the source code (exploiting well-known and robust solutions), and abstracting them back to the abstraction level of the model. Similarly to EvoMBT, an existing test generator for Java source code (in both cases EvoSuite) is exploited, but, in EvoMBT, the model itself is defined in Java. Instead, the approach in other tools is completely different, for example, Modbat and GraphWalker perform random walks on the model in order to obtain the abstract test cases, while ASMETA ATGT exploits the model checker counterexample generation.

The Table 3.1 summarizes the presented tools and CREATest, focusing on the modeling formalisms and on the format of input and output of each tool.

Table 3.1: A non-exhaustive comparison of a brief selection of available MBT tools along with the presented tool.

Tool	**Modeling formalism**	**Models (input) format**	**Abstract test cases (output) format**
EvoMBT	Extended finite-state machines	EFSM as a Java class implementing the given interfaces	List of instances of a given Java class
GraphWalker	Directed graphs	Graphical model or JSON or GraphML textual notation	Path over the directed graph stored in a file (e.g. in JSON format)
Modbat	Extended finite-state machines	EFSM implemented in a DSL built over Scala	Test cases in JUnit format or in a standalone format
Spec Explorer	Model programs (based on abstract state machines)	A model program defined in a .NET language along with a behavioral description defined using Cord	C# files consumable by a .NET framework
Qtronic	UML state machines	Graphical model or textual notation in Qtronic Modeling Language	TTCN-3 test scripts (or scripts in other test scripting languages) and HTML documentation
ASMETA ATGT	Abstract state machines	ASM implemented in AsmetaL language	ASM scenario in AVALLA language
CREATest	itemis CREATE statecharts	Graphical model	Test cases in SCTUnit language

Chapter 4

Process

For the development of CREATest, a new approach for the generation of abstract test cases was explored. The main idea of the approach is to use a ready-to-use test generator for source code. It is therefore necessary to concretize the input model into source code and to abstract the generated concrete test cases into test cases at the same abstraction level of the input model. The input models of CREATest are CREATE statecharts, the target language is Java, and the output are SCTUnit test classes. However, the basic process described here can be generalized and applied in various scenarios where the goal is to generate abstract test cases for a given model formalism. The basic process, shown in Figure 4.1, consists of three steps:

Step 1. Translate the input model into a target programming language;

Step 2. Generate test cases for the source code implementation of the model;

Step 3. Translate the generated test cases into test cases for the input model.

The process works as intended if the output of the first translation step is a one-to-one mapping of the input model and if the second step produces meaningful test cases for the translated source code. When these conditions are met, the third step must perform a simple translation and the generated test cases produced by the integration of these three steps will be meaningful. To better understand why one-to-one mapping in step 1 and meaningful test cases in step 2 are the most important properties to achieve, let's consider the raising of incoming events in SCTUnit test

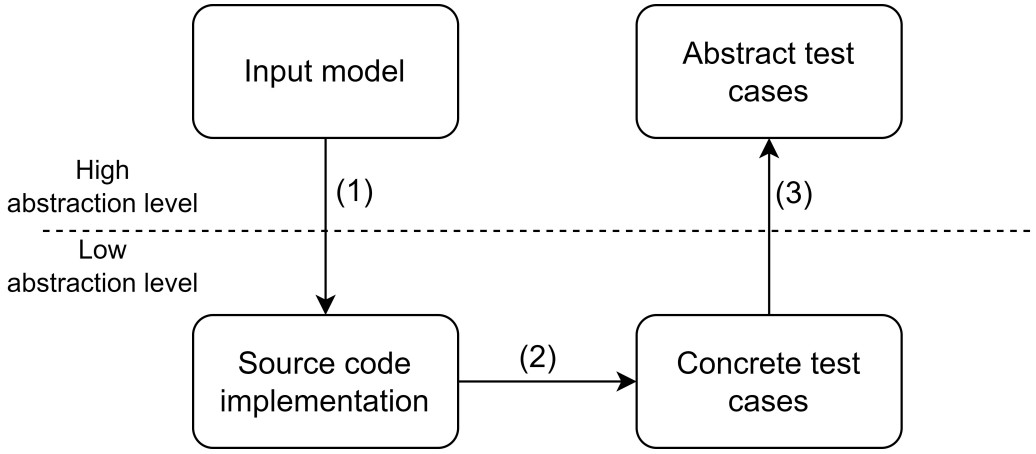

Figure 4.1: General process for abstract test cases generation.

cases. For each incoming event that can be raised, there must be a public member, such as a method, in the translated Java class (one-to-one mapping). To achieve high coverage of the Java class, the generated test cases must call these methods properly (meaningful test cases). At this point, the third step consists in translating each method call into the corresponding raise statement. In this way, the generated abstract test cases correctly raise incoming events.

Depending on the input model and the chosen target language, there may exist available tools and solutions that perform these three steps. If these tools exist for all three steps the development process consists only of integrating the existing solutions (which can be non-trivial). Otherwise, for the steps for which no tool is available, an ad-hoc solution must be implemented from scratch. For CREATest, the first two steps are covered by two well-known tools. Step 1 is done by the Java code generator integrated in itemis CREATE, whose output is a one-to-one mapping of the input model (see subsection 2.2.2). Step 2 uses the well-known EvoSuite tool, one of the best tools for JUnit test generation (see Section 2.3). There are no available solutions for translating JUnit test cases into SCTUnit test cases, therefore a solution that does this task has to be be implemented from scratch. Figure 4.2 shows the basic process shown in Figure 4.1, refined with the actual tools used in CREATest. The three steps are:

Step 1. Use the itemis CREATE Java code generator to translate the input CREATE statechart into a Java class;

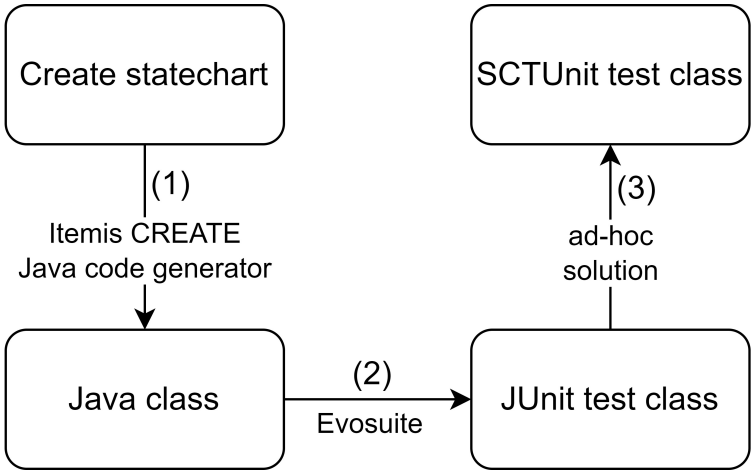

Figure 4.2: The general process adapted to CREATest.

Step 2. Use the EvoSuite tool to generate JUnit test class for the Java class;

Step 3. Translate the JUnit test class into an SCTUnit test class executable over the input CREATE statechart.

The actual implementation of the tool required some additional refinements to this basic process in order to make it work and produce meaningful test cases. The next sections delve deeper into the process implemented in CREATest.

4.1 CREATest process

Some issues encountered during the development of CREATest made the process not as straightforward as Figure 4.2 shows. The main issues are:

- The itemis CREATE Java code generator requires an SGen model in order to work. To reduce the effort required for the user to use the tool, this SGen model must be generated automatically;

- It is not enough to look at the generated JUnit test cases to generate the SCTUnit test cases. It is also necessary to collect additional information about the statechart, such as the names of its states. The subsection 4.1.1 provides a breakdown of this issue, along with the solution adopted;

- It is necessary to help EvoSuite generate JUnit test cases by providing a modified version of the Java class as input in order to obtain more meaningful test cases. The subsection 4.1.2 provides a breakdown of this issue, along with the solution adopted.

Therefore, the resulting process consists of the following steps:

Step 1. Generate an SGen model for the input CREATE statechart. The generation of the SGen model requires the statechart name, retrieved directly from the CREATE statechart, and the location where to put the generated code, provided by the user;

Step 2. Use the itemis CREATE Java code generator to generate the Java class that implements the input CREATE statechart. The code generator needs as input the statechart and the SGen model obtained in step 1;

Step 3. Generate a new Java class by modifying the Java class obtained in step 2. This new Java class is called the simplified Java class.

Step 4. Use the EvoSuite tool to generate a JUnit test class for the simplified Java class;

Step 5. Generate the SCTUnit test class starting from the JUnit test class obtained in step 4. Additional information is required to generate the SCTUnit test class. This information is: statechart, states, events and interfaces names, which are retrieved directly from the CREATE statechart, and information about the proceed times of timed events, which is retrieved from the Java class generated in step 2;

The process starts with an optional step that is required to manage namespaces. The itemis CREATE statechart language allows namespaces to be definied in the definition section of a statechart. Namespaces can be used to qualify references to statechart elements but are cumbersome to manage. For example, word that are keywords in the itemis CREATE environment, such as `event`, or in the Java environment, such as `this`, are allowed but lead to conflicts and errors in the generated artifacts. The additional step is:

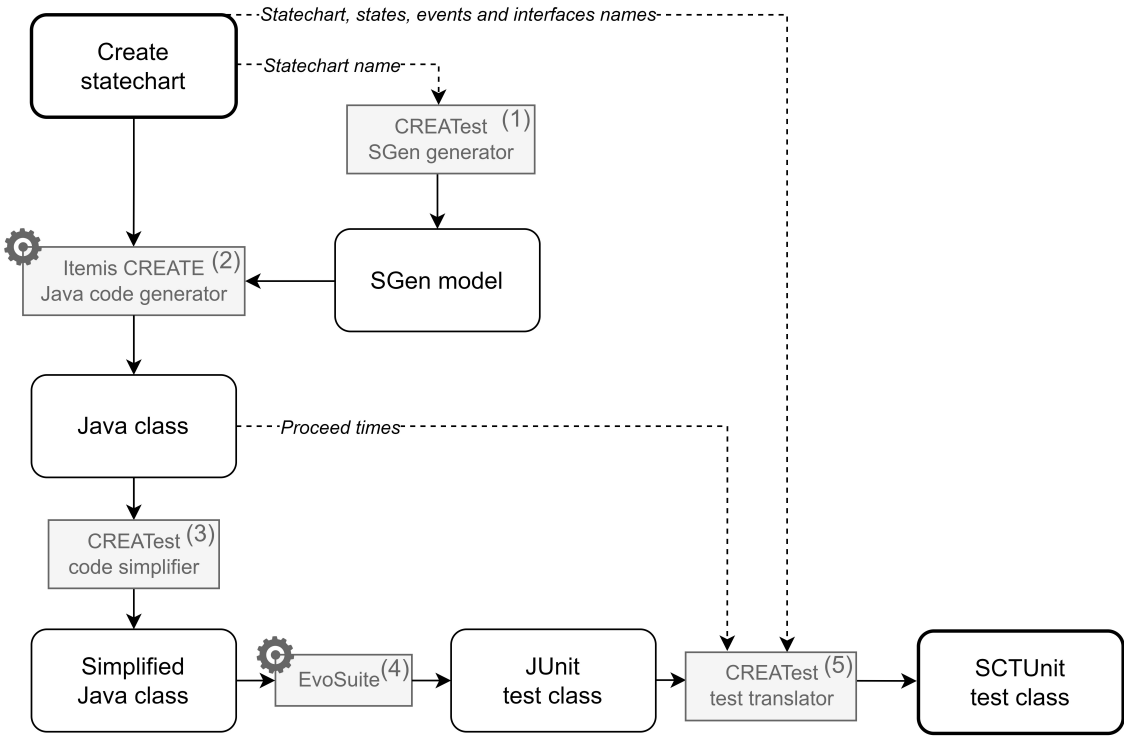

Figure 4.3: The detailed process of CREATest.

Step 0. If a namespace is defined in the definition section of the input CREATE statechart, create a new identical CREATE statechart with the namespace definition removed from the definition section. This artifact is considered the new input CREATE statechart.

This solution works, but result in the generation of an additional artifact that is almost identical to the original one. For this reason, a more elegant solution should be investigated.

The illustrated process is shown in Figure 4.3. Step 0 is only part of the process in a few cases and it is not very relevant in the overall process. Therefore, it is not shown in the diagram. Another omission is the user input required to generate the SGen model. The user input is omitted because the diagram focuses on the artifacts. The rectangles with rounded corners represent the input artifact (CREATE statechart) and the artifacts generated by the process. The input artifact (CREATE statechart) and the output artifact (SCTUnit test class) are distinguished from the others by means of a thicker outline. The gray rectangles with straight corners represents the components that carry out the process. The number shown in the upper right corner

indicates the step of the process for which the component is responsible. A gear in the upper left corner indicates that the component is an external tool. If the gear is not present, the component has been implemented ad-hoc to perform the task. The solid lines indicate the main flow of the process, i.e. which are the main artifacts needed to create a new artifact. The dashed lines indicate where the generation of an artifact requires additional information from other artifacts.

4.1.1 Collecting additional information

As Figure 4.3 shows, additional information are collected from the input CREATE statechart and from the Java generated by the Java code generator. Two kind of information are collected from the statechart:

- the name of the statechart;
- the names of states (along with their hierarchy), events and interfaces.

Just one kind of information is retrieved from the Java class:

- the mapping between temporal event IDs and the associated proceed time.

From the CREATE statechart

Before discussing why it is necessary to collect information from the statechart, a brief overview of how it is collected is presented.

A CRATE statechart is stored in XMI format in a file with the .ysc extension. Therefore, it is possible to use an XML parser to obtain the DOM object representation and collect data from it. The useful information is located in the subtree with root in the only node in the XMI with the `sgraph:Statechart` tag. This node has the `name` attribute, the value of which is the name of the statechart. Regions are nodes with the `regions` tag and have the `name` attribute, where the value is the name of the region. States and pseudo-states are nodes with the `vertices` tag and have the `xsi:type` attribute, whose value is the type of the state. Some values of this attribute are: `sgraph:State`, `sgraph:Entry`, `sgraph:FinalState`, and `sgraph:Synchronization`. State nodes with the attribute `xsi:type` equal

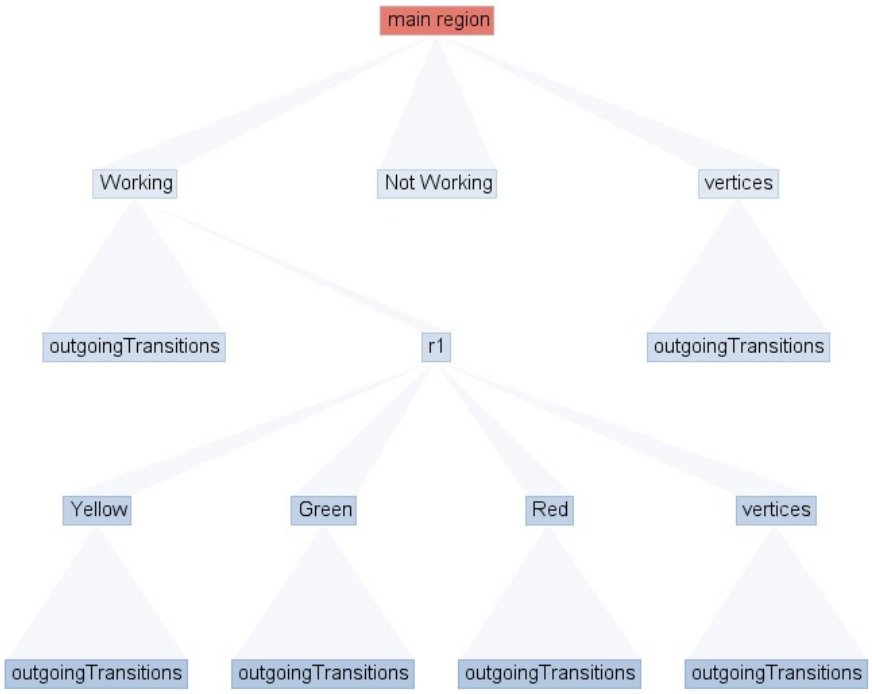

Figure 4.4: A graphical representation of the subtree with root in the node `sgraph:Statechart` tag for the SimpleTrafficLight statechart.

to `sgraph:State` also have a `name` attribute, the value of which is the name of the state. Transitions are nodes with the `outgoingTransitions` tag and have the `specification` attribute, whose value represents the transition reaction with the "trigger [guard] / effect" syntax. The useful information is the trigger, which is a comma-separated list of events with the syntax "interface.event". The interface is only present if the incoming event is defined in a named interface. Figure 4.4 shows the visualization of the subtree with root the `sgraph:Statechart` node for the SimpleTrafficLight statechart used in Section 2.2. Once the structure of the XMI representation of the statechart is known, it is possible to reconstruct the hierarchical name of each state and the names of events and interfaces, along with the statechart name.

The statechart name is a very necessary piece of information. The SGen model and the SCTUnit test class must refer to the statechart by its name and the Java code generator assigns the name of the statechart to the generated Java class. Although it is usually the same as the name of the file containing the statechart, this is not always

the case and therefore it must be directly collected. Regarding states, events and interfaces, itemis CREATE claims that "*the generated code is always correct, at least in the sense that it is a true one-to-one mapping of your statechart*". This is indeed true, but it is also true that these mappings are only one-way: given an element of the statechart, it is possible to know how it is translated into members of the generated Java class, but, given a member of the generated Java class, it could be generated starting from different elements of the statechart. For example, two events named "myEvent" and "MyEvent" both result in a `raiseMyEvent()` method. Similarly, two states named "StateA" and "STATEA" in the same top-level region named "main region" both result in the `MAIN_REGION_STATEA` enum constant. The same applies to named interfaces, translated into inner classes which always start with an uppercase letter. The problem with the states is also related to their hierarchy. In the generated Java class the enum constant relative to a state is obtained from the entire hierarchy of the state. The problem is that states and regions are separated by the underscore, but the underscore can also be used directly in the name of regions and states. Also, spaces and other characters that are allowed in the itemis CREATE environment but not in Java enums are replaced by the underscore. Therefore, it is not possible to know a priori whether an underscore represents the separation between an element and the next one in the hierarchy, or whether it is just part of the name of an element.
These issues have two effects:

- If a statechart defines two elements that are mapped to two equal Java members, the resulting Java class will have some compilation errors.
- Given a JUnit test suite, like the one produced by EvoSuite in the fourth step of the process, it is not possible to determine which are the names of the states, events and interfaces that the enum constants, methods and inner classes refer to.

The first issue must be solved by the user by refactoring the statechart in order to avoid compilation errors in the Java class. An example of the second issue is shown in Figure 4.5. The figure shows two clearly different statecharts. The enum in the

Java class generated with the statechart on the left as input is:

```
public enum State {
    MAIN_REGION_STATE_A,
    MAIN_REGION_STATE_A_REGION_1_STATE_B,
    MAIN_REGION_STATE_A_REGION_1_STATE_B
        _REGION_2_STATE_C,
    $NULLSTATE$
};
```

The enum in the Java class generated with the statechart on the right as input is:

```
public enum State {
    MAIN_REGION_STATE_A_REGION_1,
    MAIN_REGION_STATE_A_REGION_1_STATE_B
        _REGION_2_STATE_C,
    $NULLSTATE$
};
```

These two enums share a constant (besides `$NULLSTATE$`), so the following JUnit test case can be used for both the Java classes:

```
@Test
public void test() {
    Statechart stc = new Statechart();
    stc.enter();
    assertTrue(stc.isStateActive(Statechart.State.
        MAIN_REGION_STATE_A_REGION_1_STATE_B
        _REGION_2_STATE_C
    ));
}
```

But the corresponding SCTUnit test cases for the two statecharts are different, for the statechart on the left:

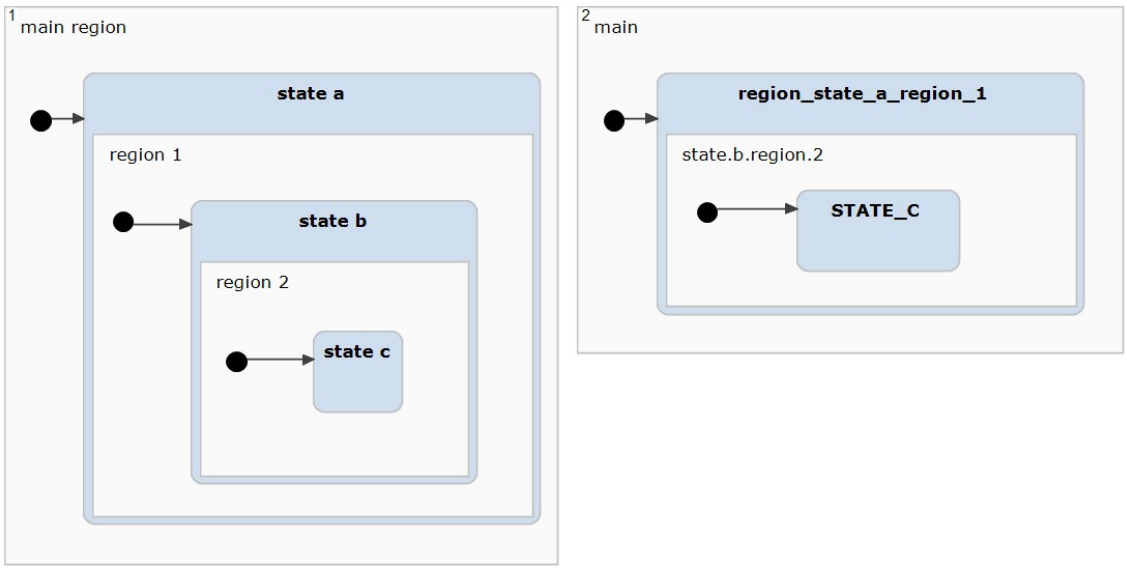

Figure 4.5: Two different statecharts that result in two Java classes with a common constant in the state enum.

```
@Test
operation test() {
    enter
    assert active(main_region.state_a.region_1.state_b.region_2.state_c)
}
```

For the statechart on the right:

```
@Test
peration test() {
    enter
    assert active(main.region_state_a_region_1.state_b_region_2.STATE_C)
}
```

As mentioned above, the implemented solution is to parse the CREATE statechart as an XMI file to obtain the names of states (with their full hierarchy, needed to refer to a state), events and interfaces as they are used in SCTUnit. For each of these elements, the corresponding member in Java is obtained, creating a mapping from the Java member to the statechart element. In the context of a single statechart, this mapping is unique because two elements that result in the same member in Java

would result in compilation errors that must be resolved by the user. Therefore, it is possible to uniquely determine to which element of the statechart a member of the Java class refers.

From the generated Java class

The reason why it is necessary to retrieve information from the Java class is to manage timed events, such as after and every. Timed events can be used on both transitions and states (entry and exit actions). If at least one timed event is used in the statechart, the generated Java class will implement the `ITimed` interface, which exposes the `raiseTimeEvent(int eventID)` and the `setTimerService(ITimerService timerService)` methods. The client code must provide an `ITimerService` implementation to the state machine by calling its `setTimerService()` method before entering the state machine. itemis CREATE provides two implementations of `ITimerService`: `TimerService` and `VirtualTimer`. The interface `ITimerService` exposes two methods:

- `setTimer(ITimed callback, int eventID, long time, boolean isPeriodic)`: the state machine calls this method to start a timer for the given `eventID` (the ID is only used in the context of Java, there is no equivalent in itemis CREATE). The `time` parameter specifies the time in milliseconds until the timer expires. When the timer expires, the timer service calls the `raiseTimeEvent(int eventID)` on the callback specified by the `callback` parameter, usually the state machine itself. The parameter `isPeriodic` is `false` for the time events of type after, `true` for the time events of type every.

- `unsetTimer(ITimed callback, int eventID)`: the state machine notifies the timer service to unset the timer associated with the `eventID`.

So, for every after and every timed event specified in the state machine, there will be a `setTimer()` call in the generated Java class, more precisely, it will be in the private methods that implement the actual behavior of the state machine.

The JUnit test cases generated by EvoSuite when the input is a class that implements the `ITimed` interface change depending on which implementations of

`ITimerService` are available in the class path given as input to EvoSuite (i.e. which binaries are in the class path). It is therefore difficult to generate meaningful SCTUnit test cases without knowing a priori which binaries will be available to EvoSuite. However, it has been found that if there is no binary relative to classes that implement the `ITimerService` interface in the class path, EvoSuite will call the `setTimerService()` method using a mock as parameter. Then EvoSuite will call the `raiseTimeEvent()` method directly within the test cases. In such a situation, it is easy to translate each `raiseTimeEvent()` method call to the corresponding SCTUnit `proceed` statement, simply by finding the link between the `eventID` used as parameter in the JUnit test case and the actual time to proceed in the SCTUnit test case. These links are obtained by reading the `setTimer(ITimed callback, int eventID, long time, boolean isPeriodic)` method calls called in the Java implementation of the state machine and looking at the second and third actual parameters used.

Note that to achieve the situation described, the CREATest tool does not generate the `TimerService` class and hides the `VirtualTimer` from EvoSuite (by deleting the corresponding .class file). It is the task of the user to prevent other binaries related to classes implementing `ITimerService` from being present in the class path.

The Table 4.1 shows the four timed events used in the CREATE statechart example shown in Figure 4.6. An event in the CREATE statechart (first column) results in a `setTimer()` call in the generated Java class (second column) and possibly in a `raiseTimeEvent()` call in some of the JUnit test cases generated by EvoSuite (third column). Parsing the Java class to find the calls to `setTimer()` allows a link to be made between an event ID and the relative proceed time. Therefore, if `raiseTimeEvent()` is called in a JUnit test case, the relative `proceed` statement to add in the SCTUnit test case (fourth column) is known. The Listing 4.1 shows a simple JUnit test case generated by EvoSuite for the Java class implementing the statechart shown in Figure 4.6. Listing 4.2 shows the SCTUnit test case derived from it.

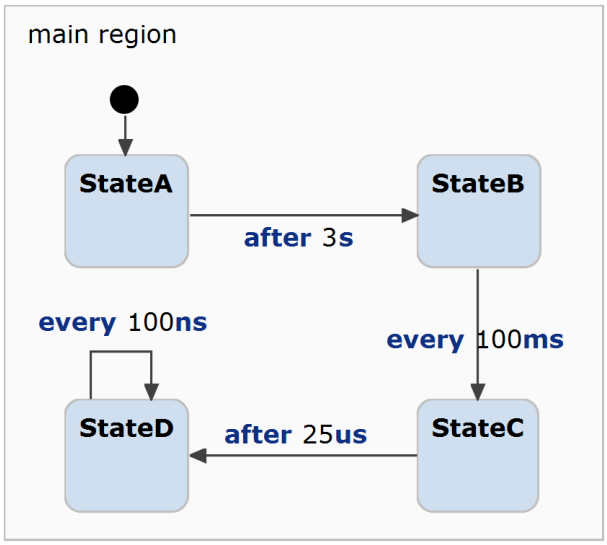

Figure 4.6: A simple CREATE statechart with four timed events.

Table 4.1: Four timed events examples (relative to the statechart shown in Figure 4.6) with the relative method calls and the final proceed statement.

CREATE statechart	**Generated Java class**	**JUnit test case**	**SCTUnit test case**
after 3s	setTimer(this, 0, (3l*1000l), false)	raiseTimeEvent(0)	proceed 3s
every 100ms	setTimer(this, 1, 100l, true)	raiseTimeEvent(1)	proceed 100ms
after 25us	setTimer(this, 2, (25l/1000l), false)	raiseTimeEvent(2)	proceed 25us
every 100ns	setTimer(this, 3, (100l/1000000l), true)	raiseTimeEvent(3)	proceed 100ns

```
@Test(timeout = 4000)
public void test08() throws Throwable {
    StatechartSimplified statechartSimplified0 = new StatechartSimplified();
    ITimerService iTimerService0 = mock(ITimerService.class, new
        ViolatedAssumptionAnswer());
    statechartSimplified0.setTimerService(iTimerService0);
    StatechartSimplified.State statechartSimplified_State0 =
        StatechartSimplified.State.MAIN_REGION_STATEC;
    statechartSimplified0.enter();
    statechartSimplified0.raiseTimeEvent(0);
    statechartSimplified0.raiseTimeEvent(1);
    boolean boolean0 =
        statechartSimplified0.isStateActive(statechartSimplified_State0);
    assertTrue(statechartSimplified0.isActive());
    assertTrue(boolean0);
}
```

Listing 4.1: Example of JUnit test case generated by EvoSuite for the statechart shown in Figure 4.6.

```
@Test
operation test08 () {
    enter
    proceed 3s
    proceed 100ms
    assert is_active
    assert active (Statechart.main_region.StateC)
}
```

Listing 4.2: SCTUnit test case obtained by translating the JUnit test case shown in Listing 4.1.

4.1.2 Helping the test generator

The original idea was to use the output of the Java code generator directly as input to EvoSuite. However, the resulting test cases were not very meaningful. For a JUnit test case to be meaningful, it is necessary that the SCTUnit test case obtained in the next step shows some properties. First, the SCTUnit test case should have no compilation errors. Second, it should pass (all test cases generated by EvoSuite pass, so the SCTUnit test cases should also pass). Even if these conditions hold, it is not guaranteed that the test case is meaningful. A meaningful test case should test some behavior of the statechart under test, for example, it should enter the state machine, raise some incoming events and check which state is active. The test suites generated by EvoSuite for the original Java class result in SCTUnit test cases that present different problems: some have errors, others fail, are useless or even empty. To improve the quality of the test cases generated by EvoSuite, a modified version of the original Java class is used as input. The idea is to reduce the visibility of members that should not be used in the test cases (either because there is no equivalent in SCTUnit or to reduce the search space of EvoSuite).

It is important to note that the reduction of the visibilities described below improves the overall quality of the generated SCTUnit test classes, as the research question RQ4 in Chapter 6 shows. However, feeding EvoSuite with a Java class with stricter visibilities does not completely solve the problem of test cases that fail, block, have compilation errors or are not generated (see the research question RQ2 in Chapter 6 for an overview of the elements in a CREATE statechart that lead to such problems). Furthermore, even with a reduced search space it is still possible to have empty or not meaningful SCTUnit test cases, especially for complex statecharts (see the research question RQ3 in Chapter 6).

A member should not be used in a test case for two reasons. The first is that there is no corresponding statement or expression to be called in SCTUnit for that member. The second reason concerns members that do have a corresponding SCTUnit statement. The problem is that as the number of elements in a statechart grows, its Java class easily becomes complex and EvoSuite struggles to work with it (the search space becomes too large). Therefore, EvoSuite should be limited to working

with members that allow significant assert statements and improve the coverage of the final SCTUnit test case. The visibilities used in the original Java class are public, protected and private. It appears that EvoSuite uses members with public and protected visibilities, so any members that should not be used in the generated test cases must be forced to be private in the new Java class. The Java class obtained by changing visibilities is called simplified Java class because it should be simpler for EvoSuite to generate meaningful test cases for it instead of the original class.

The changes relate to both fields and methods. Starting with methods, all protected methods are set to private. All the protected methods have no corresponding statements in SCTUnit. Some of these methods are associated with internal events, outgoing events, operations and typed events (both incoming and outgoing). Others, such as `getIsExecuting()`, `setIsExecuting()`, and `setStateConfVectorPosition()`, are used within the class to implement the actual behavior of the state machine. For example, `setIsExecuting(boolean value)` modifies the value of the private variable `isExecuting` and may result in an ambiguous behavior if called before the `enter` method. If a `setIsExecuting()` method call is ignored when translating from JUnit to SCTUnit, the SCTUnit test case may fail. The `setOperationCallback()` and `setTimerService()` methods are kept public even if they have no corresponding statement in SCTUnit because they are necessary for the setup of the class in JUnit test cases. Methods defined in an interface (i.e. the methods relative to operations) are kept public to avoid compilation errors. Other public methods have a corresponding statement or expression in SCTUnit but are made private. This choice is driven by the need to reduce the search space of EvoSuite. The SCTUnit test cases generated by the whole process should have a high coverage and call significant `assert` statements. The `assert` statements about whether a state is active or not, whether the state machine is active or not, and whether the active state is final or not are considered the most meaningful. The ability of the tool to generate test cases with oracles for the variables is also a desired feature. To achieve this feature, the get methods relative to the variables should be kept visible to EvoSuite. At the moment, they are kept private and an analysis should be performed to understand if the assertions

Table 4.2: Visibility changes made to the original Java class to help EvoSuite

Methods and fields from the original Java class		**Changed visibility**
Public fields	→	Private
Protected fields	→	Private
Public methods defined in interfaces	→	Public
Public methods starting with `set` and a parameter of type `long`, `double`, `boolean` or `String`	→	Private
Public methods starting with `set` and a parameter of any other type	→	Public
Public methods starting with `get`	→	Private
Protected methods	→	Private

generated by EvoSuite regarding the variables are meaningful and if the increase in the search space does not affect the quality of the generated test cases. From a first analysis, it seems that the assertions relative to variables generated by EvoSuite typically concern their initial value, which is not very meaningful. Therefore, for the moment, all the getter and setter methods relative to SCTUnit variables and constants defined in the statechart are set to private, as well as all public methods relative to outgoing events and operations. Note that in itemis CREATE the sub machines are treated like variables, but it is mandatory to keep the set methods for the sub machines public, otherwise EvoSuite will struggle to produce test cases with high coverage.

For fields, the change is straightforward: all the public and protected fields are set to private. As a result, it is not possible to directly access any field in the JUnit test cases. This choice may seem drastic but in fact almost all fields, such as those relative to variables and events, are already private. The only fields that are not private are those relative to named interfaces and constants. Each named interface is translated into an inner class and a protected variable of the inner class type. These variables are initialized in the constructor of the enclosing class and should

not be changed within a test case. As for constants, they are translated into static final fields, that are set to private (as well as the relative get methods).

All visibility changes described here are simply obtained as shown in Table 4.2. For all the methods that do not match with the one in the table, the visibility is not changed.

It may seem that these changes do not greatly affect the result of the overall process. Making members that have a corresponding statement in SCTUnit private may even seem counterintuitive. Chapter 6 presents some experimental results, including a comparison between the complete process and the process without this step. The results show that helping EvoSuite by providing an input class modified only in the visibility of its members has a huge impact on the final result.

4.2 Abstract test cases generation

The third step of the process shown in Figure 4.1 is the translation of concrete test cases into abstract test cases. In CREATest, as shown in Figure 4.2, the concrete test cases are JUnit test cases generated by EvoSuite for the class generated by the itemis CREATE code generator. The abstract test cases are SCTUnit test cases that are executable over the input CREATE statechart. Thanks to the additional steps of providing a simplified class as input to EvoSuite and collecting additional information from the statechart and the original Java class, the final translation step is trivial.

The input to this translation step is a Java class generated by EvoSuite containing different JUnit test cases. The output is an SCTUnit test class, contained in a .sctunit file. The input class containing the JUnit test cases is parsed and each method (test case) it contains is analyzed. A method (test case) is ignored if it contains a try catch statement, otherwise, a corresponding test case is added to the output SCTUnit test class. An output test case is populated translating the method call expressions into SCTUnit statements in the order in which they appear. The translation of some method calls into SCTUnit statements requires to know the additional information about the names of states, events, interfaces, the name of

Table 4.3: Mapping between JUnit method calls and SCTUnit statement.

JUnit method		**SCTUnit statement**
`enter()`	→	`enter`
`exit()`	→	`exit`
`triggerWithoutEvent()`	→	`triggerWithoutEvent`
`runCycle()`	→	`proceed 1 cycle`
`raiseTimeEvent(0)`	→	`proceed 100ms`
`raiseMyEvent()`*	→	`raise myEvent`
		`raise myInterface.myEvent`
`raiseMyTypedEvent(25L)`*	→	`raise myTypedEvent: 25`
		`raise myInterface.myTypedEvent: 25`
`assertTrue(expr)`**	→	`assert is_active`
		`assert is_final`
		`assert active(Stc.region.state)`
`assertFalse(expr)`**	→	`assert !is_active`
		`assert !is_final`
		`assert !active(Stc.region.state)`

*this method can be called either on the main class (first row) or from one of its inner classes (like `MyInterface` in the second row).

**`expr` is a boolean expression and it can be a call to the `isActive()` method (first row), a call to the `isFinal()` method (second row), or a call to the `isStateActive` method, like `isStateActive(Stc.State.REGION_STATE)` (third row).

the statechart, and the information about the proceed times. The name of the statechart is also required in the header of the test class. Table 4.3 shows the mapping between method calls and SCTUnit statements. All methods must be called from an instance that implements the statechart, or from an an instance of its inner classes where stated. In the table, the object whose method is called is omitted. It is also possible to see that the additional information has been used for all the methods except the first four. It is possible for the actual parameter of the `assertTrue` and `assertFalse` methods to be a boolean variable. In this case,

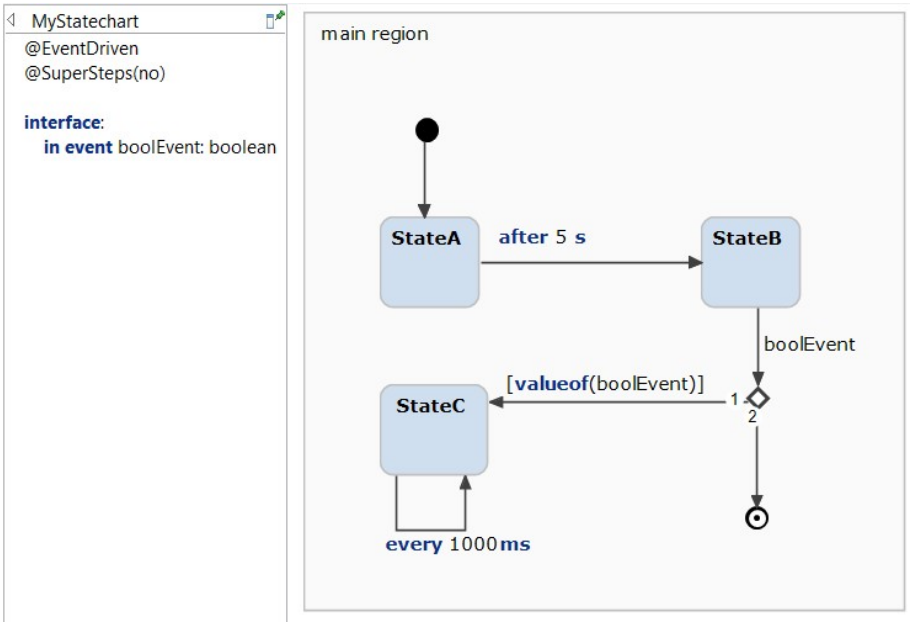

Figure 4.7: A simple statechart used to show how JUnit test cases are translated into SCTUnit test cases.

the boolean expression (i.e. the call to a method that returns a boolean value) is retrieved by parsing the variable declaration expression in the test case. The same applies to the `isStateActive` method and its parameter. It is sufficient to translate only the method calls, as everything else can be ignored. It should be necessary to translate the conditional statements and loop statements, available in the SCTUnit language, but it seems that the JUnit test cases generated by EvoSuite do not use if, while, do while, or for statements.

Now that the process has been fully described, it is possible to show a simple example of the generated test cases. Figure 4.7 shows a simple statechart with any special meaning. Based on the statechart, the itemis CREATE Java code generator generates the Java class. From this class, the simplified version with modified visibilities is obtained. Evosuite then generates a JUnit test suite for this simplified class. The coverage of the JUnit test suite is 94% and it consists of 30 test cases. An example of one of the generated JUnit test cases is:

```
@Test(timeout = 4000)
public void test10() throws Throwable {
    MyStatechartSimplified myStatechartSimplified0 = new
        MyStatechartSimplified();
    ITimerService iTimerService0 = mock(ITimerService.class, new
        ViolatedAssumptionAnswer());
    myStatechartSimplified0.setTimerService(iTimerService0);
    myStatechartSimplified0.enter();
    myStatechartSimplified0.raiseTimeEvent(0);
    myStatechartSimplified0.raiseBoolEvent(true);
    MyStatechartSimplified.State myStatechartSimplified_State0 =
        MyStatechartSimplified.State.MAIN_REGION_STATEC;
    boolean boolean0 =
        myStatechartSimplified0.isStateActive(myStatechartSimplified_State0);
    assertTrue(boolean0);
}
```

Listing 4.3: Example of JUnit test case generated by EvoSuite for the statechart shown in Figure 4.7.

With the mappings shown in Table 4.3 and the additional information retrieved from the statechart and the original Java class, the following SCTUnit test case is obtained from the previous JUnit test case:

```
@Test
operation test10 () {
    enter
    proceed 5s
    raise boolEvent: true
    assert active (MyStatechart.main_region.StateC)
}
```

Listing 4.4: SCTUnit test case obtained by translating the JUnit test case shown in Listing 4.3.

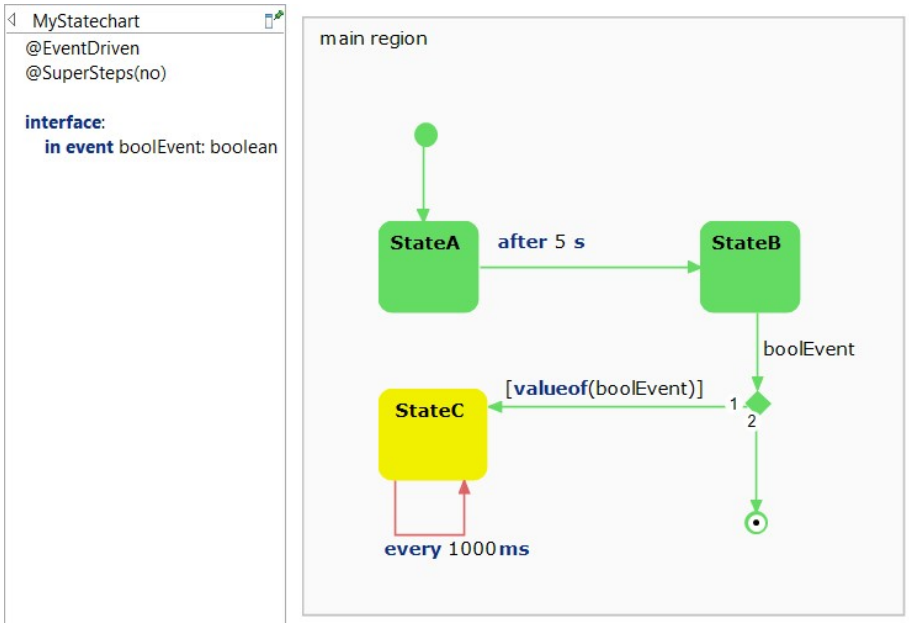

Figure 4.8: The statechart at Figure 4.7 covered by an SCTUnit test class generated by CREATest.

The entire SCTUnit test class consists of 23 test cases and it achieves an overall coverage of 91% . The covered statechart is shown in Figure 4.8.

Chapter 5

Implementation

This chapter gives an insight into how the process presented in Chapter 4 is implemented in the CREATest tool. Currently, CREATest is only available as a JAR file. Appendix A describe how to obtain and use this JAR file.

CREATest is entirely implemented in the Java language. Java has been chosen as the programming language because it is the main language used to develop the itemis CREATE infrastructure and the EvoSuite tool. In addition, Java has a robust standard library and comes with a rich ecosystem of tools, libraries and frameworks that facilitate the development of Java applications. It also makes it possible to write platform-independent applications. The IDE used to implement the tool is Eclipse (also chosen because itemis CREATE is an Eclipse-based application). The whole CREATest tool is contained within a single Maven project. Maven is a software project management and comprehension tool for Java. In this work, it is used to manage dependencies and to build the JAR. The two main tools used in CREATest are the itemis CREATE Java code generator, used headless through the scc.bat script file, and EvoSuite, whose JAR is imported in the project in order to directly invoke the method that generates the JUnit test cases. Apart from these dependencies and the standard libraries, CREATest exploits two external tools: JavaParser and StringTemplate.

JavaParser [20] is the most popular parser for the Java language. It is an open-source project that comes with a comprehensive book [25] that introduces the library, how it works and how to use it. JavaParser provides an abstract syntax tree (AST)

for the input Java code. The AST allows to both read and write a Java class programmatically and with ease. JavaParser has been used inside CREATest to parse the Java class generated by the itemis CREATE code generator in order to collect information about timed events (the mapping between each timed event ID and its proceed time, see subsection 4.1.1), to obtain the simplified Java class by changing the visibility of some members of the original class (see subsection 4.1.2), and to parse the JUnit class generated by EvoSuite to collect information about the test cases (see Section 4.2).

StringTemplate [23] is a Java template engine that allows the generation of various artifacts such as source code, web pages, emails, or any other formatted text output. The principle that StringTemplate aims to enforce is the separation of the code that handles business logic and data processing from the code that controls how information is displayed. StringTemplate is used inside CREATest to generate the SGen model (required to generate the Java class that implements the statechart) and the final SCTUnit test class. Templates defined with StringTemplate must be contained in a file a .stg extension and are very compact. For example, the template used for the generation of SCTUnit test classes is contained in only 26 lines.

The following sections present a brief overview of the software architecture, some metrics about its structure, and the results of the unit testing phase.

5.1 Software architecture

The software architecture of CREATest is described by mean of a static view and a dynamic view. The dynamic view is provided by a UML package diagram (see Figure 5.1). The static view is provided by a UML sequence diagram (see Figure 5.2).

A package diagram shows how a project is structured into packages and the dependencies between them. The package diagram shown in Figure 5.1 also shows the most relevant classes contained in each package. All the code is contained in the `createst` package. The `Createst` class is shown as a separate element because it is the glue of the project: it imports all other sub-packages and contains the main method, that calls the code contained in those sub-packages. Apart from the

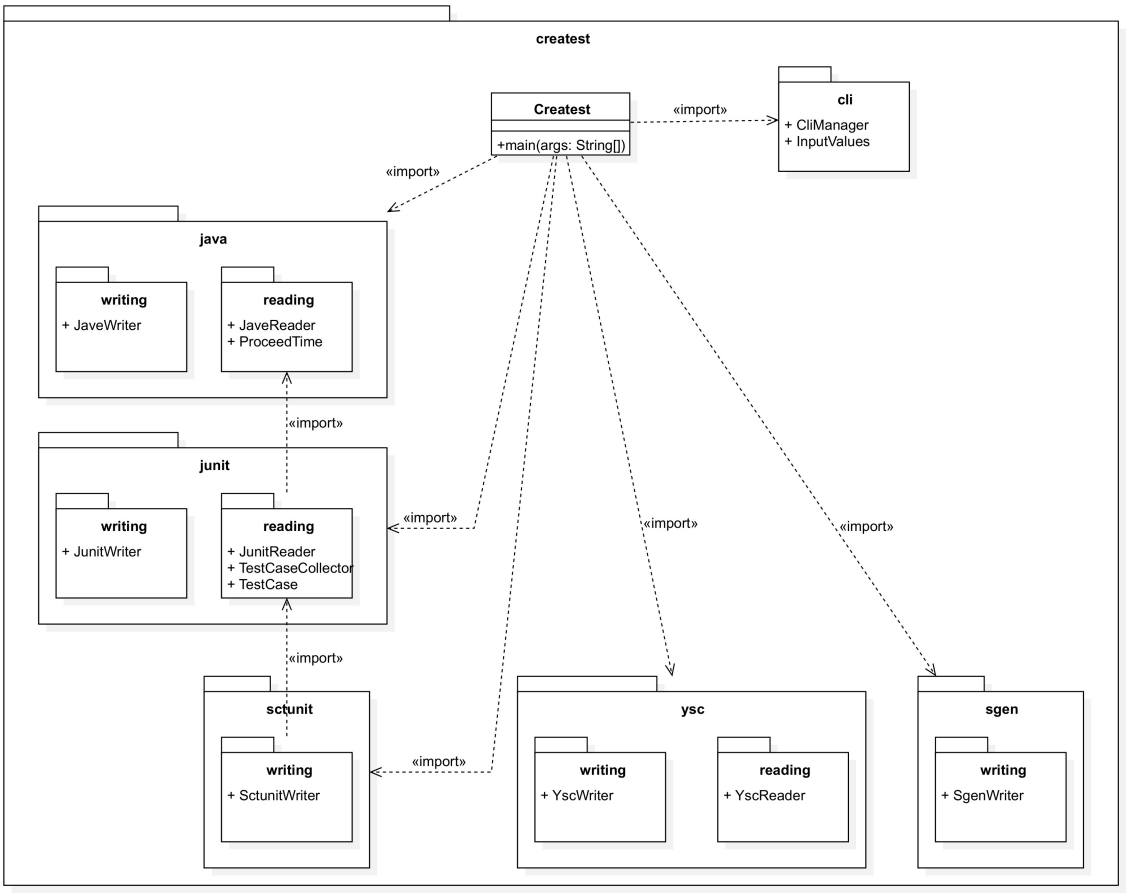

Figure 5.1: UML package diagram.

`createst.cli` package, which contains the code that implements the command line interface, all the other packages contain code that works with external artifacts and are subdivided according to the language of these external artifacts. Each of these sub-packages is further divided into the `reading` and/or `writing` packages. The `reading` packages exploit parsers (the JavaParser for Java and JUnit artifacts) in order to collect information from the artifacts. The writing packages exploit external tools (itemis CREATE code generator and EvoSuite) or the tools presented above, JavaParser and StringTemplate, in order to produce new artifacts. For example, the aim of the code contained in the `createst.java.reading` package is to work with Java classes, in particular, it will access Java classes in order to collect information from them. Similarly, the code contained in the `createst.java.writing` package is responsible for creating new Java classes. Note that technically JUnit test cases are contained within Java classes, but the code that is responsible for reading and

writing this particular type of Java class is grouped together in the `createst.junit` package.

Sequence diagrams are used to model the interaction between objects. The focus of sequence diagrams is the exchange of messages between objects and the temporal order in which they occur. The sequence diagram shown in Figure 5.2 shows the interaction (in the form of method calls) between the main method of the `Createst` class and the other classes. To make the diagram easier to read, the parameters of the method calls are omitted. The lifelines of the classes responsible for reading tasks (along with the `CliManager` class) are visually separated from the lifelines of classes responsible for writing tasks: the lifelines of writing classes are slightly further away from the `Createst` lifeline. Note that the main method of `Createst` interacts with one class from each package. The process shown produces five to six artifacts. Starting with the optional generation of a new CREATE statechart (generated only if the input statechart defines a namespace), the process then generates an SGen model, three Java classes (two classes implementing the state machine but with different visibilities and one class implementing the JUnit test suite) and a SCTUnit test class (the final output of the tool). None of these artifacts is destroyed by the process.

The two diagrams presented here provide a basic understanding of the idea behind the software architecture, without going into too much detail. In summary, the code is divided according to its functionalities. Since most of the process consists of reading and writing artifacts, the functionalities are strictly related to the type of artifacts being read and/or written. The packages are sufficiently decoupled and the glue of the whole process is the `Createst` main method, which is responsible for calling, in the right order and with the correct parameters, all the methods exposed by the rest of the code.

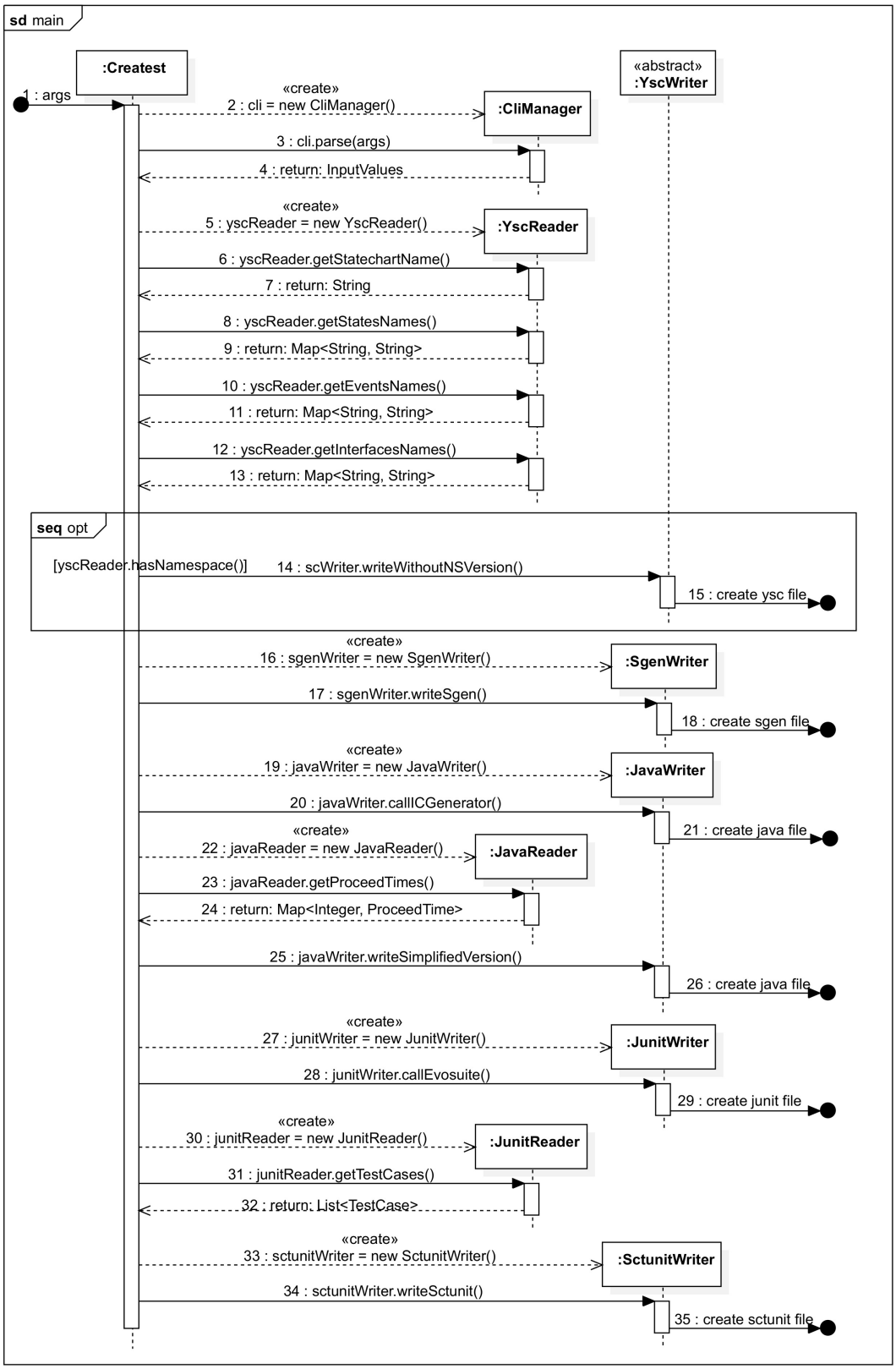

Figure 5.2: UML sequence diagram of the main method.

5.2 Metrics

During the development phase and before the testing phase, the structural properties of the software have been analyzed. The aim of the structural analysis of the software is to improve the quality of the code without changing its functionality by looking at some structural metrics.

For the structural analysis of the code, the well-known tool Structure101 [32] and the JDepend plugin for Eclipse were used. Structure101 makes it possible to visualize and manage the architecture of software systems in order to understand them better. The focus of Structure101 is on the complexity of code, which is caused by fat and tangled elements. Directly from the Structure101 documentation: "*Fat is the degree to which an item exceeds a size threshold and is applied at every level of the hierarchy. Fat is measured as Cyclomatic Complexity at the method level, and the analogous measure of the number of edges in the dependency graph is used at all other levels.*". Cyclomatic complexity, developed by Thomas McCabe, is a metric that measures the complexity of a method by counting its decision points and it is computed using the control flow graph [9]. Mathematically, the cyclomatic complexity M of a program is computed as $M = E - N + 2P$, where E is the number of edges in the control flow graph, N is the number of nodes, and P is the number of connected components. For methods, $P = 1$ and so $M = E - N + 2$. As for the tangle metric, it increases when there exist cyclic dependencies between packages. Cyclic dependencies increase the effort required to release, develop and test packages independently, so they should be avoided at all costs.

Structure101 does not provide information about the abstractness and instability of packages. Therefore, structural analysis was also carried out through the use of the JDepend plugin for Eclipse. Abstractness A is defined as $A = N_a/(N_a + N_c)$, where N_a is the number of abstract classes or interfaces in the package, and N_c is the number of concrete classes in the package [26]. Instability I is defined as $I = C_e/(C_e + C_a)$, where C_e is the efferent coupling (number of classes in other packages on which classes in the package depend), and C_a is the afferent coupling (number of classes in other packages which depend on classes within the package) [26]. If $I = 0$ or

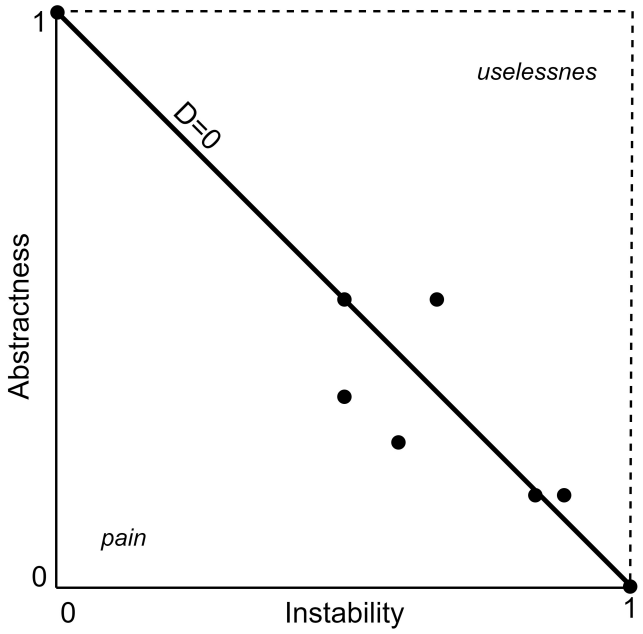

Figure 5.3: Abstractness vs Instability graph.

close to zero, the classes in the package depend on nothing but are heavily depended upon (they should be stable, changes affect a lot of other packages). If $I = 1$ or close to one, the classes in the package depend on many classes outside it, but classes in other packages are not dependent on them (they can be unstable, changes do not affect other packages). Both instability and abstractness varies between 0 and 1. These two metrics must be read together with the distance measure D: $D = |A + I - 1|$, D is a balance between A and I and should be zero or very close to zero. Abstractness, instability and distance are typically represented in the Abstractness vs Instability graph, with the abstractness on the y-axis and instability on the x-axis. The region in the graph near to the $(0, 0)$ point ($D = 1$) is known as the *zone of pain*. The region near to the $(1, 1)$ point ($D = 1$) is known as the *zone of uselessness*. The corner cases where $D = 0$ are $(0, 1)$ (package with concrete and unstable classes) and $(1, 0)$ (package with abstract and stable classes).

The analysis carried out with Structure101 showed that there were no cyclic dependencies in the project (no tangles), but two methods were fat. Some refactoring was done in order to reduce the cyclomatic complexity of these two methods. On the other hand, the use of JDepend showed that all packages had abstractness equal

Table 5.1: Results of the JDepend analysis on the final version of the software.

Package	**Ca**	**Ce**	**A**	**I**	**D**
createst	0	10	0.00	1.00	0.00
createst.cli	1	1	0.33	0.50	0.16
createst.java.reading	2	3	0.25	0.60	0.14
createst.java.writing	1	5	0.16	0.83	0.00
createst.junit.reading	1	7	0.20	0.87	0.07
createst.junit.writing	1	1	0.50	0.50	0.00
createst.sctunit.writing	1	1	0.50	0.50	0.00
createst.sgen.writing	1	1	0.50	0.50	0.00
createst.ysc.reading	1	2	0.50	0.66	0.16
createst.ysc.writing	1	0	1.00	0.00	0.00

to zero and in some cases not very high instability (i.e. high distance). A complete refactoring of the structure of the packages, that lead to the structure shown in the package diagram of Figure 5.1, and the addition of some abstract classes and interfaces significantly improved the quality of the software architecture. The values of C_a, C_e, A, I and D after the refactoring are shown in the Table 5.1 and in Figure 5.3.

5.3 Unit testing

After the refactoring phase, the entire software was tested at the unit level. The testing framework used was JUnit. The testing phase was quite challenging because almost all implemented methods involve reading or writing external files, so the input space can be huge. A single test class was implemented for each package. The aim was to achieve a high coverage, including testing corner cases such as null input and empty paths or wrong paths as input. The `callICGenerator()` method of the `JavaWriter` has not been tested due to the difficulties associated with testing the call to the scc.bat file, which requires a full installation of itemis CREATE in order to work and which position is not known a priori. Consequently, also the main method

Table 5.2: Coverage achieved with unit testing.

Package	**Coverage [%]**	**# Covered Instructions**	**# Missed Instructions**
createst	0.00	0	562
createst.cli	95.7	649	29
createst.java.reading	92.7	153	12
createst.java.writing	78.4	269	74
createst.junit.reading	97.1	868	26
createst.junit.writing	96.0	96	4
createst.sctunit.writing	100.0	40	0
createst.sgen.writing	100.0	46	0
createst.ysc.reading	99.0	513	5
createst.ycc.writing	96.9	126	4

of the `Createst` class, that calls the `callICGenerator()` method of the `JavaWriter` class, has not been tested either. However, these methods, as well as all the others, have been subjected to rigorous and extended manual testing. The overall coverage of the implemented JUnit test cases is 79.4%. Excluding the untested code, the total coverage rises to over 90%. A breakdown of the coverage and the number of covered instructions is given in the Table 5.2. Due to the extensive manual testing performed during the development of the software, the final automated JUnit test cases did not reveal any severe issue. Apart from a few minor bugs that has been fixed, the testing phase also highlighted the need to improve the exception handling.

Chapter 6

Experiments

This chapter presents the experiments conducted to evaluate the performance of the tool. Specifically, the experiments are designed to answer the following research questions (RQs):

- **RQ1**: Is the tool able to generate SCTUnit test cases that pass?
- **RQ2**: What are the elements in the CREATE statechart that cause a generated test case to fail, block, have compilation errors, or not be generated?
- **RQ3**: How does the tool perform in terms of coverage and quality of the generated SCTUnit test cases for different levels of statechart complexity?
- **RQ4**: Does helping the test generator by providing a simplified input improve the quality of the generated SCTUnit test classes?
- **RQ5**: Does the coverage obtained by EvoSuite affect the coverage of the final SCTUnit test class?

The following sections provide a detailed overview of the process by which these questions were answered.

6.1 Data collection

In order to address the research questions, particularly RQ2, it is necessary to feed the tool with a large number of CREATE statecharts as input. For this reason, it is

not sufficient to create some statecharts manually. It was decided to use all the CREATE statecharts (file with .ysc extension and XML format) available on GitHub. The statecharts on GitHub were divided into three groups: the statecharts available in all official itemis CREATE repositories (`https://github.com/itemisCREATE`), the statecharts available in the STL4IoT folder in the mde-tmu/STL4IoT repository (`https://github.com/mde-tmu/STL4IoT/tree/main/STL4IoT`), and the statecharts from all the other repositories. The initial number of statecharts in each group was 68, 34 and 97 respectively, for a total of 199 statecharts. The division into three groups was done to facilitate the automation of the experiments process described in Section 6.2, but is not relevant for the results analysis presented in Section 6.3. STL4IoT is a library of statechart templates for designing IoT systems composed of atomic CREATE statechart components that model heterogeneous aspects of IoT systems, such as sensors, actuators, network, and controller. A comprehensive overview of the repository is available at [27].

Before running CREATest on all the statecharts obtained from GitHub, a manual inspection of each statechart was performed. The inspection was necessary to remove statecharts with errors or not suitable as input for the tool. From the itemis CREATE group 15 statecharts were removed: 9 because they used the C/C++ domain, 1 because it used the SCXML domain, and 5 because they were almost identical to other statecharts in the same group. From the STL4IoT group 5 statecharts were removed because they were almost identical to other statecharts in the same group. Finally, 46 statecharts were removed from the last group: 7 because they used the C/C++ domain, 3 because they contained compilation errors, 15 because because they were almost identical to other statecharts in the same group, 15 because they were too simple or meaningless (e.g. the default CREATE statechart generated when a new .ysc file is created), 3 because they caused a runtime exception that does not allow them to be visualized, and 3 because they were already in the itemis CREATE group. In the end there were 133 statecharts left.

If two statecharts in the same group have the same name, the artifacts generated for the first statechart are overwritten by the ones generated for the second statechart. Furthermore, if a statechart imports other statecharts, it is necessary to

run the tool on the imported statecharts first and on the importing statechart afterward. For these reasons, it was necessary to change the names of the statecharts to avoid overwriting and to change the names of the files so that the execution in lexicographic order runs first on the imported statecharts and then on the importing ones. In addition, all statechart names have been made equals to the corresponding file names.

6.2 Process automation

In order to answer RQ4, it was necessary to slightly modify the tool. The process described in Section 4.1 and shown in Figure 4.3 had to be adapted. After generating the simplified Java class, EvoSuite must be run on both the original class and the simplified class, and an SCTUnit test class is derived from both the EvoSuite outputs. The modified process is shown in Figure 6.1. To make the diagram clearer, the arrows representing the additional information are omitted. The artifacts derived from the original Java class are called "standard" and the artifacts derived from the simplified Java class are called "simplified", but it is important to note that they are not a simplified version of the standard one. The simplified SCTUnit class is the actual output of the tool.

Once that the modified tool was obtained, it was necessary to automate the process and run the tool one hundred and thirty-three times. A batch file was implemented to run the tool for all the CREATE statecharts contained in the directory provided as input. For each of the three groups into which the statecharts were divided (itemis CREATE group, STL4IoT group and other repositories group), the batch file was executed and the SCTUnit test classes were grouped into two SCTUnit test suites (one for the standard SCTUnit classes and one for the simplified SCTUnit classes). A SCTUnit test suite allows the execution of all the test classes it contains. Before executing the test suites, the generated SCTUnit test classes were manually inspected to check which ones contained compilation errors. The execution of the test suites made it easy to obtain the SCTUnit coverage information for both the standard and simplified test cases of all the statecharts. The tool also

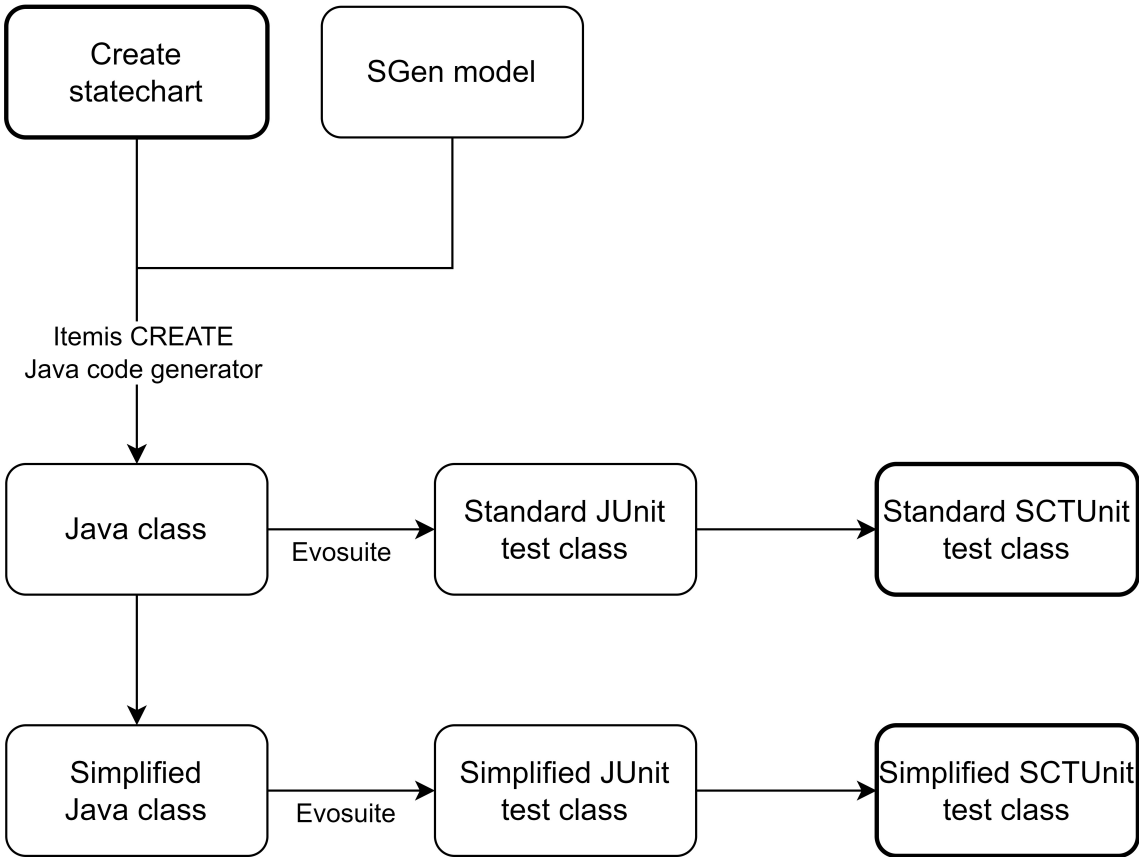

Figure 6.1: The CREATest process adapted to generate SCTUnit test cases without passing through the simplified Java class.

produces a CSV file containing the information about the coverage of the JUnit test cases generated by EvoSuite (for both the standard and simplified cases).

In addition, to answer RQ3, a script was implemented to retrieve information about the number of states, the average depth and the maximum depth of each CREATE statechart contained in an input directory and its sub-directories. The script write this data to a CSV file. The depth of a state is equal to the number of states above it in its hierarchy plus one. The maximum hierarchy, the average hierarchy and the number of states are used in this context to describe the complexity of a statechart.

All the pieces of information collected was grouped into a CSV file with the following columns:

- **Directory**: the directory where the statechart is stored. It depends on the group of the statechart (itemis CREATE group, STL4IoT group or other repos-

itories group).

- **Statechart**: the name of the statechart.
- **NumStates**: the number of states in the statechart.
- **AvgDepth**: the average depth of the states in the statechart.
- **MaxDepth**: the maximum depth of the states in the statechart.
- **StandardEvosuiteCoverage**: the coverage (between 0 and 1) of the JUnit test class generated by EvoSuite for the original Java class.
- **StandardSCTUnitCoverage**: the coverage (between 0 and 1) of the SCTUnit class derived from the standard JUnit test class.
- **StandardSCTUnitStatus**: the status of the standard SCTUnit class. It can be PASSED, FAILED, BLOCKED, ERRORS or NOT GENERATED.
- **SimplifiedEvosuiteCoverage**: the coverage (between 0 and 1) of the JUnit test class generated by EvoSuite for the simplified Java class.
- **SimplifiedSCTUnitCoverage**: the coverage (between 0 and 1) of the SCTUnit class derived from the simplified JUnit test class.
- **SimplifiedSCTUnitStatus**: the status of the simplified SCTUnit class. It can be PASSED, FAILED, BLOCKED, ERRORS or NOT GENERATED.

6.3 Results analysis

The following is an analysis of the CSV file obtained as described in the previous section, broken down to answer the five research questions posed at the beginning of the chapter. In this analysis, the statecharts are all grouped together and no distinction between the three groups is considered anymore.

6.3.1 Results for RQ1

Out of the 133 statecharts, for 112 of them an SCTUnit test class in which all test cases passed was generated. This is almost the 84.21% out of the total. Considering the remaining 21 statecharts, for 8 (6.02%) of them the generated SCTUnit test class exhibited at least one test case that fails, for 1 statechart (0.75%) the execution of the generated SCTUnit test class freezes, for 2 statecharts (1.50%) the generated SCTUnit test class presented at least one compilation error, and for the last 10 statecharts (7.52%) the tool was not able to generate a SCTUnit test class. The reasons leading to these problems are discussed in the research question RQ2.

Given that the output of EvoSuite is non-deterministic, it follows that the final output of CREATest is also non-deterministic. Therefore, if the SCTUnit test class generated for a statechart contains test cases that fail, block or have compilation errors, it is possible to run the tool more times to increase the chance of obtaining an SCTUnit class where all test cases pass. Running CREATest several times is not a solution for the statecharts for which the tool was unable to generate the SCTUnit test class, because the execution is blocked by the same error each run. It also appears that running the tool multiple times is not a solution for the 2 statecharts for which the generated test class showed compilation errors. In fact, 20 runs were tried for each of the these two statecharts, with unsuccessful results. For the remaining 9 of the 11 statecharts for which the tool was run multiple times, a correct SCTUnit test class was obtained. For these statecharts, the number of runs required varied from 1 to 12. However, increasing the number of runs is not a consistent approach. It can be seen as a practical, albeit limited, strategy to mitigate the occurrence of failing or blocking test cases in some scenarios.

In conclusion, by trying multiple runs of the tool where necessary, a correct SCTUnit test class was generated for a total of 121 out of 133 statecharts (90.98%).

> ***RQ1**: Is the tool able to generate SCTUnit test cases that pass? Yes, the tool is able to generate SCTUnit test cases that pass for more than the 90% of the CREATE statecharts used as input.*

6.3.2 Results for RQ2

The main reasons why a generated SCTUnit class presents at least one test case that fails are the use of variables in incoming timed events (3 out of 8) and the poor design of the statechart (3 out of 8). In the first case, the problem is that CREATest is not able to reconstruct the amount of time that must be proceeded if it is not directly specified on the event that triggers a transaction. In the second case, the problem is poor statechart design that results in ambiguous behavior of the statechart that the tool is not able to catch. The last reason of failing test cases is the use of operations or variables modified by operations in transition guards. These tests fail because mock methods in JUnit are ignored by CREATest.

For only 1 statechart the execution of the generated SCTUnit class blocked. The problem is the design of the CREATE statechart, for which there is a series of event that, if raised, results in an infinite loop. A test case in the generated SCTUnit test class raised such a series of event calls. Since the problem is inherent in the design of the statechart, there is no way to prevent CREATest from eventually generating test cases that result in infinite loops and therefore in a blocked execution of the SCTUnit test class.

The use of element names that are keyword in the SCTUnit environment led to the generation of SCTUnit classes with compilation error for 2 statecharts. This keywords are handled in itemis CREATE by forcing the circumflex character (^) as prefix in the SCTUnit environment. CREATest is not able to understand when the circumflex should be used, therefore some compilation errors could occur.

In the presented experiments, for 10 of the 133 CREATE statechart the tool was unable to generate any SCTUnit class. For 2 of them the reason was that they defined regions without names. If the name of a region is not specified, itemis CREATE automatically generates a new name and assigns it to the region, but CREATest does not know what this name is. For 1 statechart the problem was the use of the character "ü" in the name of a state. This character, like the other accented letters, is not handled correctly by itemis CREATE and leads to a compilation error in the generated Java class. For 1 statechart the problem was a conflict in the name of the statechart and a keyword in Java. When such a conflict occurs, the itemis

Table 6.1: Sources of problems for generation of the SCTUnit test class

Problem	**Source of the problem**	#
Failing test cases	Use of variables in incoming timed events	3
	Poor design of the statechart	3
	Use of operations or variables modified by operations in transitions guards	2
Blocking execution	Series of event raised in a generated test case that results in an infinite loop on the statechart	1
Test cases with errors	Use of names in the statechart that are keywords in SCTUnit	2
Test class not generated	Use of sub machines that defines outgoing events	6
	Use of unnamed regions	2
	Use of characters (such as ü) not allowed in Java	1
	Use of a Java keyword as the statechart name	1

CREATE code generator adds the suffix "SM" to the name of the generated Java class (and thus to the name of the file), therefore CREATest is unable to find it. The main problem that causes CREATest to fail when attempting to generate an SCTUnit test class concerns sub machines and affects 6 of the 133 statecharts (60% of the statecharts for which a SCTUnit class was not generated). These statecharts import sub machines that define at least one outgoing event. This results in a method declaration that is not handled correctly by CREATest when the simplified Java class is obtained. Table 6.1 presents a summary of the number of occurrences of each problem according to its sources of origin.

***RQ2**: What are the elements in the CREATE statechart that cause a generated test case to fail, block, have compilation errors, or not be generated? The main elements are: timed events that use variables, operations, names with special characters, names that conflict with Java or SCTUnit keywords, and sub machines that define out events.*

6.3.3 Results for RQ3

The approach taken to answer this question is to group the statecharts into clusters representing different levels of complexity. Since the focus of this question is the coverage of the generated SCTUnit class, only statecharts for which an SCTUnit class was generated without compilation errors are considered. Therefore, both passing and failing SCTUnit classes are included in the analysis. The number of such statecharts is 120 out of the total of 133. The features used for the clustering are:

- The number of states in the statechart.
- The average depth of the states in the statechart.
- The maximum depth of the states in the statechart.

The use of three features also allows the result of the clustering to be graphically visualized. Three different well-known clustering algorithms were tried: K-means, DBSCAN and OPTICS. For all the algorithms, the implementation in Python provided by the machine learning library scikit-learn [29] was used. Unlike K-means, DBSCAN and OPTICS do not need to know the number of clusters in advance. In K-means, it was necessary to try different reasonable values of K (number of clusters) to find the best value. Additionally, DBSCAN and OPTICS are able to identify outliers. In order to select the best algorithm with some reasonable values for its parameters, two metrics were used: the silhouette score and the Davies-Bouldin Index (DBI). The silhouette of a sample varies between -1 and 1 and it is the measure of how similar the sample is to samples in its own cluster (cohesion) compared to samples in other clusters (separation). High silhouette values are preferred. The silhouette of a clustering is the mean of the silhouettes of all its points, values above 0.5 are considered generally good and values above 0.7 are considered very good [30]. For the Davies-Bouldin Index, values are non-negative and lower values indicates better clustering results. The DBI is lowered increasing the separation of clusters and decreasing the variation within clusters.

DBSCAN achieved the best results in both silhouette and DBI. DBSCAN

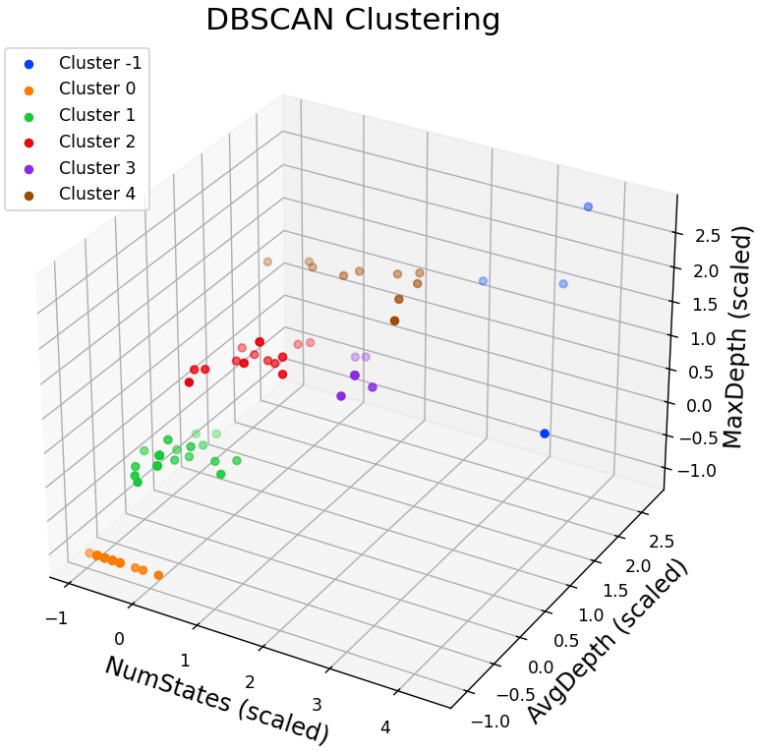

Figure 6.2: Three dimensional scatter plot of the clustering obtained by DBSCAN.

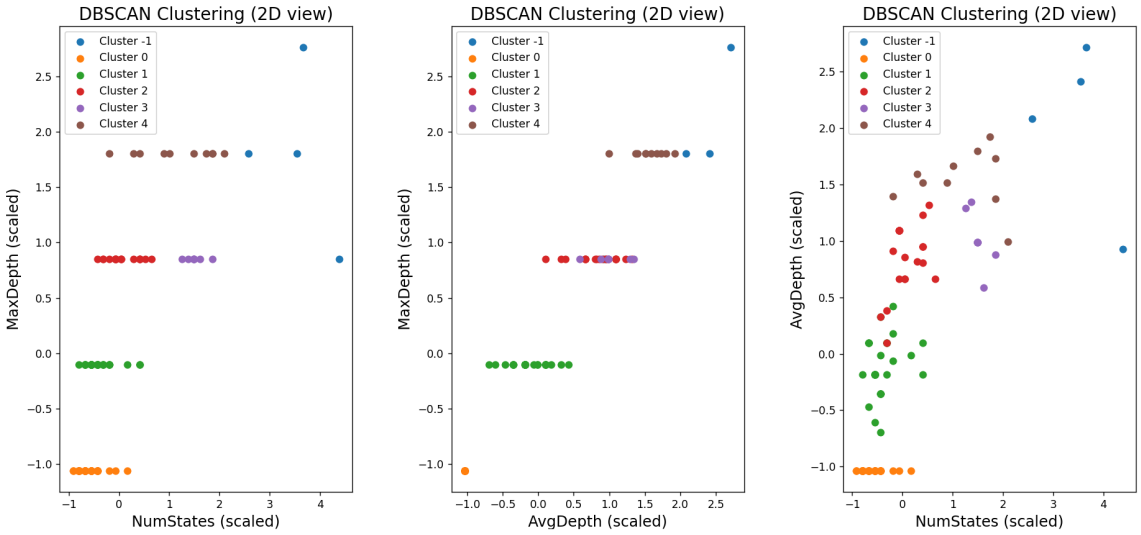

Figure 6.3: Two dimensional views of the scatter plot of the clustering obtained by DBSCAN.

(Density-Based Spatial Clustering of Applications with Noise) finds core samples of high density and expands clusters from them [11]. The parameters associated with the best values of silhouette and DBI are ϵ (or eps, the maximum distance between

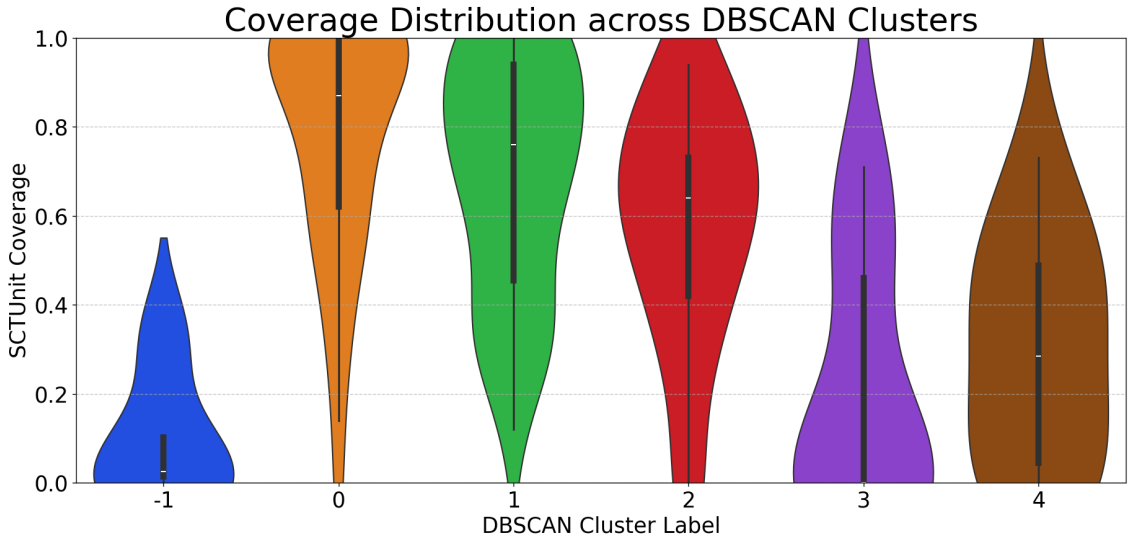

Figure 6.4: Violin plot with the coverage distribution of the clusters obtained by DBSCAN.

two samples for one to be considered as in the neighborhood of the other) equal to 0.55 and minimum number of samples (min_samples, the number of samples in a neighborhood for a point to be considered as a core point) equal to 2. The algorithm achieves a silhouette score of 0.66 and a Davies-Bouldin Index of 0.64. Five clusters and 4 outliers (cluster -1) were identified. The graphical representation in three dimensions of the result of the clustering is shown in Figure 6.2. Whereas Figure 6.3 shows the three two dimensional views. It appears clear that the main feature used to discriminate clusters was the maximum depth of the states, which is sufficient to uniquely identify the 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 the clustering.

Figure 6.4 shows the violin plot with the distribution of the coverage for each cluster. It is clear that the clusters have different coverage distributions and that the coverage decreases as the number of states and the maximum depths increase (and thus when the complexity of the statechart increases). The following list provides an overview of each cluster accompanied by an example statechart extracted from it. Figure 6.5 shows the example statecharts for clusters 0, 1, 2 and 4 and Figure 6.6

shows the example for cluster 3. It is not possible to show the statecharts labeled as outliers due to their dimension.

- **Cluster 0**: This is the most populated cluster as it contains 45 statecharts. This cluster only contains statecharts with a maximum depth of 1 and therefore an average depth of 1. Clearly, these statecharts have no composite states and the number of states is limited (the maximum is 10). It is clear that this cluster groups the least complex statecharts of the dataset. The coverage distribution is the best of all clusters, with a mean of 78.16% and a median of 87%. The number of fully covered statecharts (100% coverage) in this cluster is 20, which is almost the 45% of the samples in the cluster and almost the 72% of the total number of statechart with 100% coverage (28). Only for one statechart the coverage was 0%. This shows that the tool sometimes performs poorly even for simple statecharts. The generated SCTUnit classes are composed of meaningful test cases. The example statechart chosen for this group consists of 4 states and the achieved coverage is 92%. Only one generated test case is empty and almost all are meaningful. An example of a meaningful generated test case is:

```
@Test
operation test13 () {
    enter
    raise operate
    proceed 20s
    raise operate
    assert is_active
    assert active (alarm2.main_region.partiallyArmed)
}
```

- **Cluster 1**: This cluster contains 31 statecharts. All statecharts have a maximum depth of 2. The number of states is between 2 and 12. Therefore, the cluster contains statecharts that are still sufficiently simple. In fact, the distribution of the coverage is still good (mean equals to 68.77% and median equals

to 76%). The cluster contains the remaining 8 statecharts for which a 100% coverage was achieved and none of the generated SCTUnit classes got coverage equal to 0%. The generated SCTUnit test cases are meaningful. The example statechart selected from the cluster has 10 states and an average depth of 1.60 (more states with depth 2 than states with depth 1). The statechart is not trivial, but the tool was able to achieve 78% of coverage and generate meaningful test cases such as:

```
@Test
operation test10 () {
    enter
    raise speed_up
    raise red_light_on
    raise door_open
    assert !is_final
}
```

Only one test case was empty but the tool generated five equal and useless test cases, such as:

```
@Test
operation test33 () {
    assert !is_active
}
```

- **Cluster 2**: This cluster contains 21 statecharts with maximum depth equals to 3 and a number of states between 5 and 14. The cluster contains statecharts for which an high coverage was achieved (up to 94%) and only for 2 statecharts the coverage was 0%. The mean is 57.29% and the median is 64%. The generated SCTUnit test cases are still meaningful. The example statechart selected from the cluster has 11 states and an average depth of 2.09. The coverage achieved by the generated SCTUnit test class is 94%, only one test case was empty and the tool was able to find non-trivial test cases such as:

```
@Test
operation test46 () {
    enter
    raise toggle
    raise temp_down
    raise temp_down
    raise temp_down
    assert is_active
    assert active (AirConditioner_Unit._AirConditioner_.
        MonitorRoomTemp.MonitoringTemp.RoomTempChanging.
        TemperatureChange.CoolDown
    )
}
```

- **Cluster 3**: This cluster contains 9 statecharts. Like cluster 2, it contains only statecharts with maximum depth equals to 3. The number of states is between 19 and 24. The increase in the number of states in the statecharts of this cluster worsened the performance of the tool. The coverage distribution is the worst of all clusters (except for the outliers). The mean coverage is 20.67% and the median is 0.00%. For 5 of the 9 statecharts the coverage is 0% and the highest coverage is 71%. The generated SCTUnit classes are not very meaningful and in some cases completely useless because they contain only empty test cases or test cases with only one statement. The selected example has 20 states and an average depth of 2.4. It is the statechart with the highest coverage in the cluster (71%). It has been chosen to show that even if the coverage obtained is high, the resulting SCTUnit class is not very significant (five test cases were empty and ten had only the `assert !is_active` statement). For example, the following is one of the most significant test cases:

```
@Test
operation test14 () {
    enter
    raise Mode.warning
    exit
    assert !is_active
}
```

- **Cluster 4**: This cluster contains 10 statecharts, all with a maximum depth of 4. The number of states ranges from 7 to 26. Even if the maximum depth and the mean average depth are higher than in cluster 3, the number of statecharts with a relatively low number of states is significant (40% of the statecharts have a number of states lower than the minimum number of states in cluster 3). The coverage benefits from this and the distribution is better than that of cluster 3: the mean coverage is equal to 29.50% and the median is equal to 28.50%. The maximum coverage is 73% and only for three statecharts the achieved coverage is 0%. The chosen example is the one with maximum coverage: it has 7 states and an average depth of 2.42. It is a small but deep statechart and the generated SCTUnit class presents seven empty test cases, the others are not very meaningful except for a few, such as the following:

```
@Test
operation test09 () {
    enter
    raise off
    assert is_active
    assert active (Network_Component._Network_Component_._off_)
}
```

- **Cluster -1 (outliers)**: The algorithm has identified 4 outliers. It is clear that the outliers are the most complex statecharts in the dataset, and given their complexity, the tool struggles to obtain a good result. One of the four

Table 6.2: Comparison between the clusters obtained by DBSCAN.

Cluster	**Samples number**	**Mean NumStates**	**Mean AvgDepth**	**MaxDepth**	**Mean Coverage**
0	45	3.42	1.00	1	78.16%
1	31	5.23	1.53	2	68.77%
2	21	9.19	2.11	3	57.29%
3	9	21.11	2.20	3	20.67%
4	10	18.10	2.52	4	29.50%
-1	4	38.00	2.81	4*	9.25%

*This is a mean value.

statecharts has a maximum depth and an average depth of 3 and 2.16, therefore not very high, but the highest number of states for the whole dataset, which is 45. Its coverage is 2%. The statechart with the highest coverage (32%) is the only one in the dataset with a maximum depth of 5 and it also has 39 states. The remaining 2 statecharts have a maximum depth equals to 4 (like cluster 4), but both have a number of states and an average depth that is higher than all the statecharts in cluster 4. Their coverage is 0% and 3%. The generated SCTUnit classes for the outliers have little to no utility. For example, in the SCTUnit class generated for the statechart with 45 states 11 test cases were empty and 38 had only the `assert !is_active` statement.

Table 6.2 summarizes the characteristics of each cluster. It seems that the number of states is the main factor that make it difficult for CREATest to generate meaningful SCTUnit classes, like comparing cluster 3 and 4 suggests.

***RQ3**: How does the tool perform in terms of coverage and quality of the generated SCTUnit test cases for different levels of statechart complexity? The coverage and the quality of the generated SCTUnit test cases are really good for simple statecharts, but decrease as the number of states, the maximum depth and the average depth increase.*

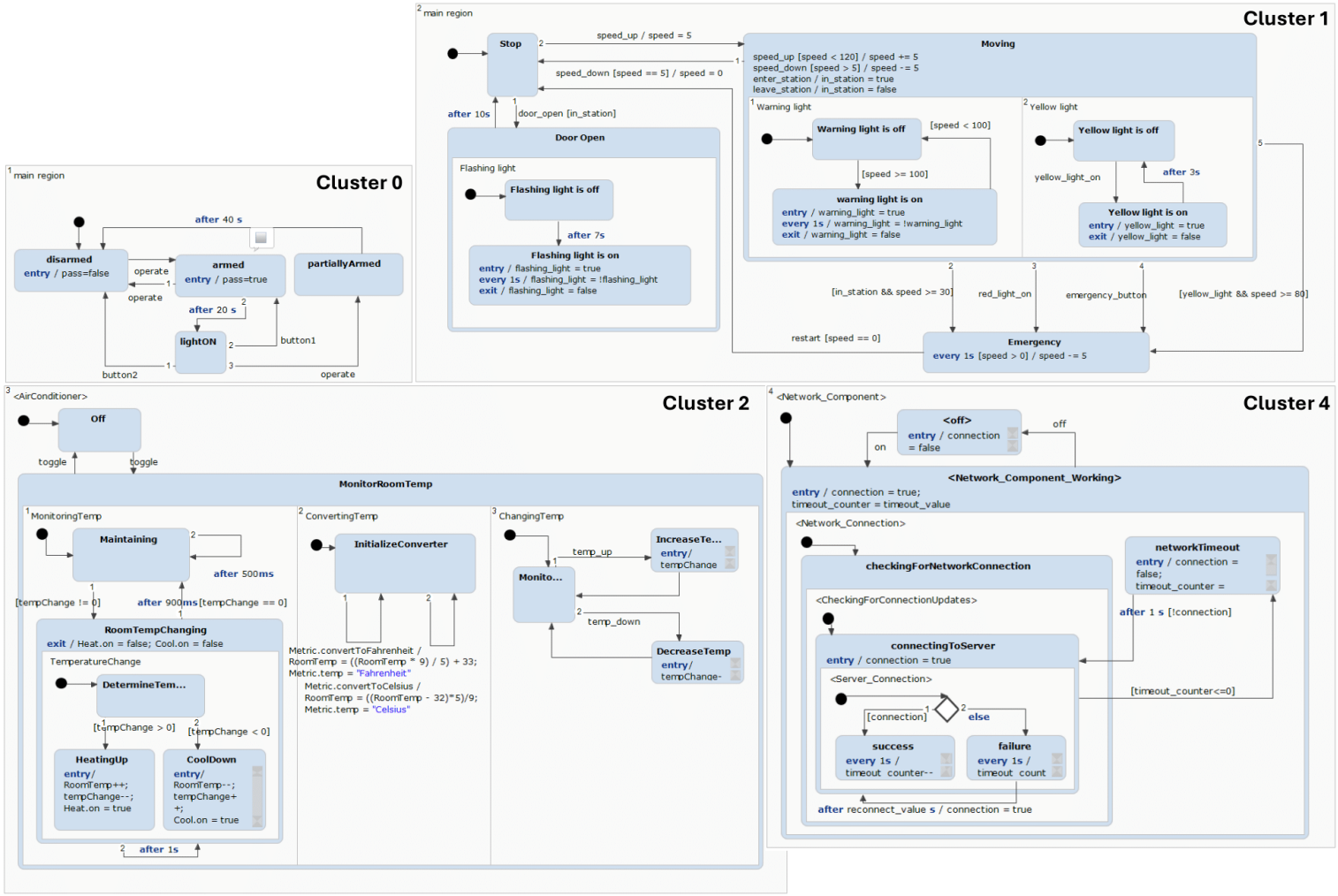

Figure 6.5: The CREATE statechart selected from clusters 0, 1, 2 and 4.

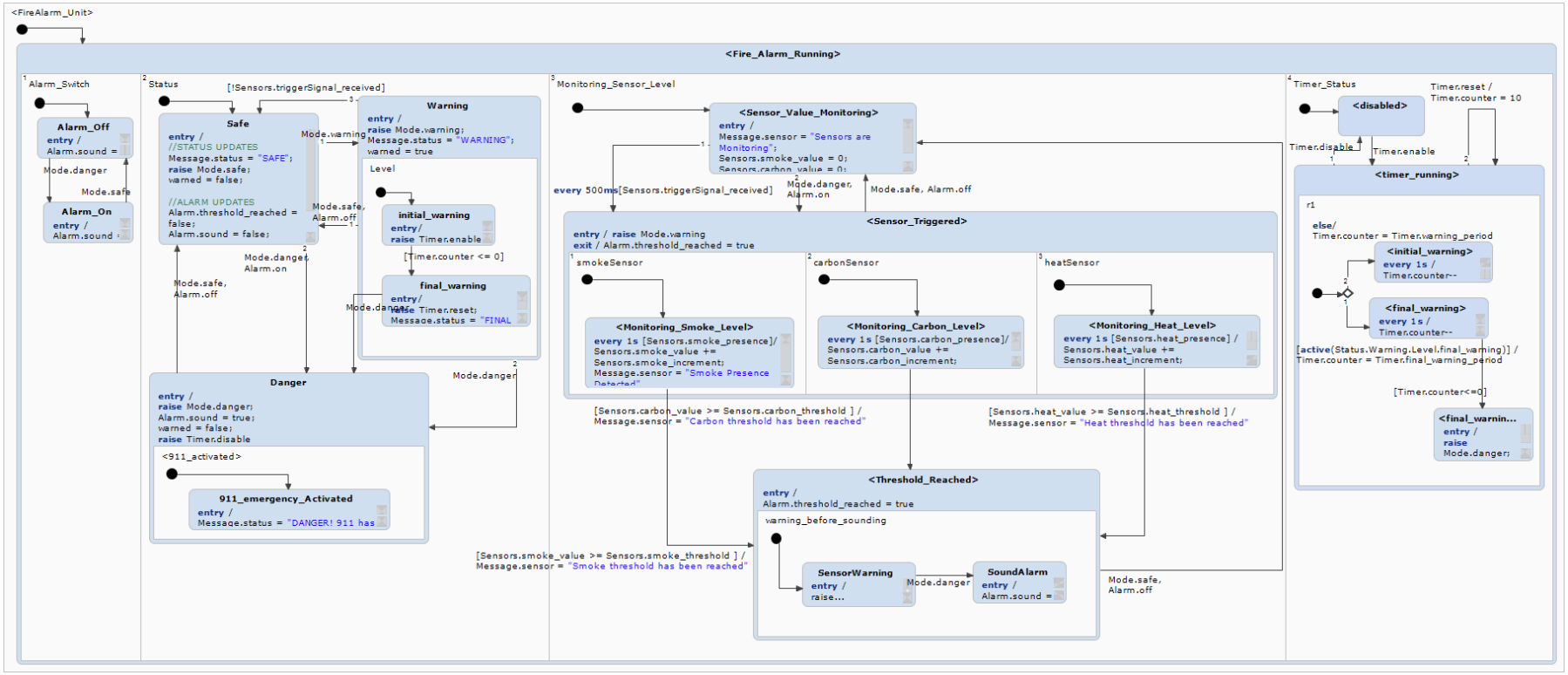

Figure 6.6: The CREATE statechart selected from cluster 3.

Table 6.3: Comparison between standard SCTUnit classes and simplified SCTUnit classes on 133 CREATE statecharts.

Metrics	**Standard SCTUnit classes**	**Simplified SCTUnit classes**
Number of generated classes	123	123
Number of generated classes that pass	82	112
Number of generated classes that fail	21	8
Number of generated classes with errors	20	2
Number of generated classes that block	0	1
Average coverage of the generated classes	16.93%	61.42%

6.3.4 Results for RQ4

Using a modified version of the class generated by the itemis CREATE Java code generator as input to EvoSuite significantly improves the overall results of CREATest. Changing the visibility of some Java members from public and protected to private reduces the search space of EvoSuite and allows to find better solutions. The experiments showed that this reduction in the EvoSuite search space increases the number of statecharts for which the tool is able to generate a SCTUnit test class that pass and leads to an overall improvement in the quality of the generated SCTUnit test classes in terms of coverage. Table 6.3 shows a comparison between the standard SCTUnit classes (generated without passing per the simplified Java class) and the simplified SCTUnit classes (the actual output of the tool, generated passing per the simplified Java class) for the 133 statecharts in the dataset. The average coverage increases by 44.49%, and the number of generated SCTUnit classes for which all test cases pass increases from 61.65% to 84.21%.

***RQ4**: Does helping the test generator by providing a simplified input improve the quality of the generated SCTUnit test classes? Yes, helping EvoSuite significantly improves the quality of the generated SCTUnit classes.*

6.3.5 Results for RQ5

The effort of EvoSuite is aimed at generating a JUnit test suite with an high coverage for the Java class given as input. In the context of this tool, the output of EvoSuite is an intermediate result. However, it would be beneficial if the effort of EvoSuite also affects the quality of the final results. In other words, it is expected that 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, and vice versa, a low JUnit coverage should mean a low SCTUnit coverage. The experiments confirm that these two metrics are strongly correlated. The Pearson correlation coefficient between the coverage of the simplified SCTUnit classes that either pass or fail and the coverage of the JUnit classes from which they are derived is 94.48%.

> ***RQ5**: Does the coverage obtained by EvoSuite affect the coverage of the final SCTUnit test class? Yes, the coverage achieved by EvoSuite is strongly correlated to the coverage achieved by the SCTUnit test class.*

Chapter 7

Limitations and Future Work

This chapter aims to discuss the limitations of both the approach and the tool presented in this thesis. The main goal of the tool, that is, the automatic generation of SCTUnit classes for CREATE statecharts, can be considered as achieved. However, there is a lot of room for improvement, both in the core functionalities of the tool and in the usability. The following list contains the main limitations of the tool and directions for future work to solve or at least limit them:

- CREATest is a command line interface (CLI) tool and it is only available as a JAR file. Command line interfaces are known to be heavy to use, especially if they have many optional and required options, which is the case of CREATest. Following the example of itemis CREATE, which was also made available as a web cloud editor, CREATest should be made available as a web service, which is more user-friendly than the CLI.

- The JAR file can only be used on Windows operating systems due to some OS dependent operations on files. JAR files should be executable on any machine where the Java Virtual Machine (JVM) is available. Therefore, some refactoring should be done in order to make CREATest executable also on Linux operating systems and macOS.

- CREATest uses EvoSuite version 1.0.6 to generate JUnit test classes. This version limits the tool to be used with Java 8. At the moment, the latest available version of EvoSuite is 1.2.0, which supports Java 9 and is based on a

preforming algorithm known as DynaMOSA. The latest EvoSuite version was not used in CREATest because it caused an exception during execution. This problem should be investigated and solved in order to use the version 1.2.0 of EvoSuite in CREATest and verify if it improves the performance of the tool. Otherwise, it should be verified if the performance of the tool can be improved by more carefully selecting the criteria used by EvoSuite 1.0.6 to generate the JUnit test class.

- The SCTUnit classes generated by CREATest sometimes contain empty test cases or groups of identical test cases. To make the generated SCTUnit classes cleaner, the empty test cases should be removed from the final result and identical test cases should be aggregated into one.

- If the input to CREATest is a CREATE statechart that defines a namespace, a new identical statechart with the namespace removed is generated and is used as new input by the tool. A more elegant solution for managing namespaces in the input statechart should be investigated.

- The unit testing done at the end of the development of the tool revealed a poor exception handling, which should be improved.

- Currently, CREATest hides the get methods relative to the variables defined in the CREATE statechart to EvoSuite. This prevents the tool from generating SCTUnit test cases with oracles for the variables. It should be investigated whether or not allowing EvoSuite to generate JUnit test cases that call these get methods and translating them into SCTUnit `assert` statements will result in more meaningful SCTUnit test cases.

- For some CREATE statecharts, CREATest is not able to generate SCTUnit test classes where all test cases pass and in some cases an error is encountered during the process and the tool does not generate any output. The main causes of these problems was identified and should be fixed:

  - The tool is unable to retrieve information from incoming timed events where the time to proceed is not directly specified in the transaction but variables are used. A solution should be investigated.
  - The part of the tool that performs the translation from JUnit to SCTUnit is not able to read mock methods, therefore some behaviors of the statecharts could not be captured, like the use of operations or variables modified by operations in the guards of the transitions.
  - The name of the statechart and the names of its elements can be keywords in the Java environment or in the SCTUnit environment. The tool should be able to understand when such cases occur and add the appropriate prefixes and suffixes where necessary.
  - The regions may be unnamed. In this case, the tool execution results in an error and the output is not generated. To solve this problem, the tool should be able to understand when a region is unnamed and which name is automatically assigned by itemis CREATE to the region.
  - The use of some special characters, such as accented letters, in the names of the elements of a statechart leads to compilation errors in the generate Java class and thus to the impossibility for the tool to generate the output. A solution to this problem should be investigated.
  - The execution of the tool results in an error if the input CREATE statechart imports a sub machine that defines outgoing events. The problem is due to a bug in the code that generates the simplified version of the Java class, which should be fixed.

- CREATest exploits the itemis CREATE Java code generator and EvoSuite to perform the first two steps of the process shown in Figure 4.2. For the third step an ad-hoc translator from JUnit to SCTUnit has been developed. This translator should be extended to work with more complex JUnit inputs and made available as standalone product.

Chapter 8

Conclusions

The aim of this thesis is to present CREATest, a tool for the automatic generation of test cases for statecharts in itemis CREATE, and the novel approach it implements. The lack of an available solution for the automatic generation of SCTUnit classes for CREATE statecharts, both as a feature of itemis CREATE and in academic research, prevented the use of the itemis CREATE toolkit as a model-based testing tool. Furthermore, an automatic test case generator for CREATE statecharts can be used to enhance the validation of models when using itemis CREATE for model-driven development or before starting the model-based testing process. CREATest aims to close this gap by providing a ready-to-use solution for the automatic generation of test cases for CREATE statecharts.

The proposed approach consists of three basic steps: the generation of a source code implementation of the model (first step), the automatic generation of test cases for the source code (second step), and the translation of the test cases to the same level of abstraction of the model (third step). The strength of the approach proved to be the exploitation of available and powerful solutions for the first two steps of the approach: the itemis CREATE Java code generator and EvoSuite. In addition, the intuition to provide a slightly modified version of the generated Java class as input to EvoSuite, in order to reduce its search space and achieve better results, proved successful. Thanks to the quality of the output of EvoSuite with a modified version of the Java class generated by the itemis CREATE code generator as input, the final step, for which an ad hoc solution was implemented, resulted in a one-to-

one translation of Java methods into SCTUnit statements. It turned out that the translation required some additional information that had to be retrieved directly from the statechart and the generated Java class in order to be completed.

The experiments conducted on a dataset of 133 CREATE statecharts showed that the tool is able to generate meaningful test cases that pass and achieve a good coverage, especially for simple statecharts. Therefore, the goal of providing a new approach for the generation of abstract test cases and at the same time filling a gap in the itemis CREATE toolkit can be considered to be achieved. However, as the complexity of the input CREATE statecharts (i.e. the the number of states and their maximum and average depth) increases, the performance of the tool decreases. If the complexity of the statechart is too high (e.g., more than 30 states and a maximum depth of 4), the tool is not able to generate meaningful test cases. Such complex statecharts are likely to be more useful in the model-driven development approach, where the models are used to derive a source code implementation, rather than for model-based testing.

While the tool has shown promising results, further enhancements and refinements are needed to improve the quality of the generated test cases and to limit the number of statecharts for which the tool is unable to produce an error-free SCTUnit class for which all test cases pass.

Bibliography

[1] Cyrille Valentin Artho et al. "Modbat: A Model-Based API Tester for Event-Driven Systems". In: *Hardware and Software: Verification and Testing*. Ed. by Valeria Bertacco and Axel Legay. Cham: Springer International Publishing, 2013, pp. 112–128.

[2] *Asmeta - Overview*. URL: `https://asmeta.github.io` (visited on 06/2024).

[3] Omar Badreddin and Timothy Lethbridge. "Badreddin, O.: A Manifestation of Model-Code Duality: Facilitating the Representation of State Machines in the Umple Model-Oriented Programming Language". In: (Jan. 2012).

[4] Michele Beretta. *UniBG LaTeX Thesis Template*. URL: `https://github.com/micheleberetta98/unibg-thesis-template`.

[5] Andrea Bombarda, Silvia Bonfanti, and Angelo Gargantini. "Automatic Test Generation with ASMETA for the Mechanical Ventilator Milano Controller". In: *Testing Software and Systems*. Ed. by David Clark, Hector Menendez, and Ana Rosa Cavalli. Cham: Springer International Publishing, 2022, pp. 65–72. ISBN: 978-3-031-04673-5.

[6] Kwang Ting Cheng and A. S. Krishnakumar. "Automatic functional test generation using the extended finite state machine model". In: *Proceedings of the 30th International Design Automation Conference*. DAC '93. Dallas, Texas, USA: Association for Computing Machinery, 1993, pp. 86–91. ISBN: 0897915771. DOI: `10.1145/157485.164585`. URL: `https://doi.org/10.1145/157485.164585`.

[7] A. Cimatti et al. “NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking”. In: *Proc. International Conference on Computer-Aided Verification (CAV 2002)*. Vol. 2404. LNCS. Copenhagen, Denmark: Springer, July 2002.

[8] *ConformIQ: AI-Powered Software Testing Automation Solutions*. URL: https://www.conformiq.com/ (visited on 06/2024).

[9] *Cyclomatic Complexity - GeeksforGeeks*. URL: https://www.geeksforgeeks.org/cyclomatic-complexity (visited on 07/2024).

[10] *Davies–Bouldin index - Wikipedia*. URL: https://en.wikipedia.org/wiki/Davies%E2%80%93Bouldin_index (visited on 07/2024).

[11] *DBSCAN*. URL: https://scikit-learn.org/stable/modules/generated/sklearn.cluster.DBSCAN.html (visited on 07/2024).

[12] Agoston E Eiben and James E Smith. *Introduction to evolutionary computing*. Springer, 2015.

[13] *EvoMBT*. URL: https://github.com/iv4xr-project/iv4xr-mbt/wiki (visited on 06/2024).

[14] Raihana Ferdous et al. “EvoMBT: Evolutionary model based testing”. In: vol. 227. Mar. 2023, p. 102942. DOI: 10.1016/j.scico.2023.102942.

[15] *Finite State Machines | Brilliant Math & Science Wiki*. URL: https://brilliant.org/wiki/finite-state-machines (visited on 07/2024).

[16] Gordon Fraser and Andrea Arcuri. “EvoSuite: Automatic test suite generation for object-oriented software”. In: Sept. 2011, pp. 416–419. DOI: 10.1145/2025113.2025179.

[17] *GraphWalker*. URL: https://github.com/GraphWalker/graphwalker-project/wiki (visited on 06/2024).

[18] David Harel. “Statecharts: a visual formalism for complex systems”. In: *Science of Computer Programming* 8.3 (1987), pp. 231–274. ISSN: 0167-6423. DOI: https://doi.org/10.1016/0167-6423(87)90035-9. URL: https://www.sciencedirect.com/science/article/pii/0167642387900359.

[19] *Itemis CREATE – Documentation Overview*. URL: https://www.itemis.com/en/products/itemis-create/documentation/ (visited on 06/2024).

[20] *JavaParser*. URL: https://javaparser.org/ (visited on 07/2024).

[21] Wenbin Li, Franck Le Gall, and Naum Spaseski. "A Survey on Model-Based Testing Tools for Test Case Generation". In: Mar. 2017, pp. 77–89. ISBN: 978-3-319-71734-0. DOI: 10.1007/978-3-319-71734-0_7.

[22] *Mealy and Moore Machines in TOC - GeeksforGeeks*. URL: https://www.geeksforgeeks.org/mealy-and-moore-machines-in-toc (visited on 07/2024).

[23] *Mealy and Moore Machines in TOC - GeeksforGeeks*. URL: https://www.geeksforgeeks.org/mealy-and-moore-machines-in-toc (visited on 07/2024).

[24] *Modbat: Model-based Tester*. URL: https://gitlab.com/cartho/modbat/-/wikis/home (visited on 06/2024).

[25] Danny van Bruggen Nicholas Smith and Federico Tomassetti. *JavaParser: Visited*. Leanpub, 2023.

[26] *Object-oriented metrics by Robert Martin*. URL: https://kariera.future-processing.pl/blog/object-oriented-metrics-by-robert-martin/ (visited on 07/2024).

[27] Clyde Rempillo and Sadaf Mustafiz. *STL4IoT: A Statechart Template Library for IoT System Design*. 2023. arXiv: 2311.18175 [cs.SE]. URL: https://arxiv.org/abs/2311.18175.

[28] Monalisa Sarma et al. "Model-based testing in industry - A case study with two MBT tools". In: May 2010. DOI: 10.1145/1808266.1808279.

[29] *scikit-learn: machine learning in Python*. URL: https://scikit-learn.org/stable/ (visited on 07/2024).

[30] *Silhouette (clustering) - Wikipedia*. URL: https://en.wikipedia.org/wiki/Silhouette_(clustering) (visited on 07/2024).

[31] *Spec Explorer*. URL: https://learn.microsoft.com/en-us/previous-versions/visualstudio/spec-explorer/ee620411(v=specexplorer.10) (visited on 06/2024).

[32] *Structure101 - Software Architecture Development Environment (ADE)*. URL: https://structure101.com/ (visited on 07/2024).

[33] Mark Utting, Alexander Pretschner, and Bruno Legeard. "A taxonomy of model-based testing approaches". In: *Software Testing, Verification and Reliability* 22.5 (2012), pp. 297–312. DOI: https://doi.org/10.1002/stvr.456. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.456. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.456.

[34] Sebastian Vogl et al. "Evosuite at the SBST 2021 Tool Competition". In: *2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST)*. 2021, pp. 28–29. DOI: 10.1109/SBST52555.2021.00012.

[35] Neil Walkinshaw, Ramsay Taylor, and John Derrick. "Inferring Extended Finite State Machine models from software executions". In: *2013 20th Working Conference on Reverse Engineering (WCRE)*. 2013, pp. 301–310. DOI: 10.1109/WCRE.2013.6671305.

[36] *What is a state machine?* URL: https://www.itemis.com/en/products/itemis-create/documentation/user-guide/overview_what_are_state_machines (visited on 06/2024).

[37] *YAKINDU Statechart Tools*. URL: https://github.com/itemisCREATE/statecharts (visited on 09/2024).

[38] Muhammad Nouman Zafar et al. "Model-Based Testing in Practice: An Industrial Case Study using GraphWalker". In: *Proceedings of the 14th Innovations in Software Engineering Conference (Formerly Known as India Software Engineering Conference)*. ISEC '21. Bhubaneswar, Odisha, India: Association for Computing Machinery, 2021. ISBN: 9781450390460. DOI: 10.1145/3452383.3452388. URL: https://doi.org/10.1145/3452383.3452388.

Appendix A

User Guide

This user guide provides information on how to install the tool, the requirements for using the tool, an overview of the command line interface, and a step-by-step guide to learn how to use the tool.

A.1 Requirements and installation

The CREATest tool is distributed as a JAR at the following GitHub repository:

`https://github.com/PellegrinelliNico/test-generator-for-yakindu`

The name of the file is CREATest-0.0.2.jar. The requirements for using the tool are:

- A machine with a Windows operating system installed and at least 32MB of free hard disk space.
- Java Development Kit (JDK) version 1.8 installed on the machine. The JDK is available for the download at `https://www.oracle.com/it/java/technologies/javase/javase8-archive-downloads.html`.
- A working Eclipse-based installation of itemis CREATE (both standalone and Eclipse plug-in are supported). The supported versions of itemis CREATE are 5.2.1 or 5.2.2 (the correct functioning of the tool is not guaranteed with different versions of itemis CREATE). itemis CREATE is available under license. No feature provided by the professional license is required to use the tool, so the standard license is sufficient. The installation of itemis CREATE comes

with an evaluation license that is valid for 30 days. Licenses are available at `https://www.itemis.com/en/products/itemis-create/licenses/`. itemis CREATE can be downloaded after filling out the form available at `https://info.itemis.com/products/itemis-create/download/`.

- A directory named *libs* containing the following dependencies as JAR files:
  - *ANTLR Runtime* version 3.3,
  - *Apache Commons CLI* version 1.6.0,
  - *EvoSuite* version 1.0.6,
  - *JavaParser Core* version 3.25.6,
  - *StringTemplate* version 4.0.2.

  The *libs* directory must be located in the same directory of CREATest-0.0.2.jar. The *libs* directory with all needed dependencies can be obtained downloading and unzipping the libs.zip archive file available at `https://github.com/PellegrinelliNico/test-generator-for-yakindu`.

A.2 CLI

The CREATest tool can be used by the user via a command line interface (CLI). The following options are required:

- `-scc`: the absolute path to the scc.bat file contained in the itemis CREATE installation.
- `-projectPath`: the absolute path to the Java project containing the input CREATE statechart. It is the base path for the other options.
- `-sourceDir`: the relative path to the directory containing the input CREATE statechart.
- `-sourceFile`: the name of the file without extension where the input CREATE statechart is stored. The extension can be either .ysc or .sct.

- `-targetPackage`: the package in dot notation where the .java files generated by the tool will be placed.

The following options are optional:

- `-h`, `-help`: print the help message, regardless of other options.
- `-binaryDir`: the relative path to the directory where the binary (.class) files will be placed. The default value, which is sufficient for standard Java projects, is "bin". For example, if the project specified in the `-projectPath` option is a Maven project, the value of this option should be "target\classes".
- `-targetDir`: the relative path to the directory that will contain the target package. If it is an existing source folder, it cannot have sub-directories. The default value is "src".
- `-evoTestDir`: the relative path to the directory where the generated JUnit tests will be placed. The default value is "evosuite-tests".
- `-evoSearchBudget`: the EvoSuite search budget, expressed in seconds. It must be a positive integer.

If an input option provided by the user is incorrect (e.g. the specified source file does not exist) or one of the required options is missing, a message describing the error is displayed.

A.3 Usage

Once all the requirements are met, it is possible to use the CREATest tool. The following is a step-by-step guide to using the tool from scratch:

Step 1. Create a new Java project in the itemis CREATE Eclipse-based installation:

- Start itemis CREATE.
- Select the workspace and click *Launch*.
- Navigate to *File -> New -> Project... -> Java Project*.

- Enter the project name and click *Finish*.
- Note: The project can be a standard Java project or any project that supports Java development, including Maven projects.

Step 2. Optionally, create a new source folder for tests:

- Right-click on the newly created Java project.
- Navigate to *New -> Source Folder*.
- Enter the source folder name and click *Finish*.

Step 3. Create a new folder for the CREATE statechart:

- Right-click on the newly created Java project.
- Navigate to *New -> Folder*.
- Enter the folder name and click *Finish*.

Step 4. Create the CREATE statechart:

- Right-click on the newly created folder.
- Navigate to *New -> Statechart Model*
- Enter the file name (with .ysc extension) and click *Next*.
- Select *Default Domain* and click *Finish*.
- If a window asking to switch the perspective appears, click on *Yes*.

Step 5. Edit the CREATE statechart:

- Open and modify the newly created statechart file as needed.

Step 6. Run the CREATest JAR:

- Open a terminal window.
- Navigate to the directory containing the CREATest-0.0.2.jar file and the *libs* directory.
- Run the CREATest JAR with the appropriate options.

Step 7. Analyze the newly created artifacts:

- Examine the generated artifacts.
- Run the SCTUnit class over the statechart.

The itemis CREATE editor should not be used during the execution of CREATest.

After the first five steps, the workspace should look like this:

```
C:\path\to\workspace
└── MyProject
    ├── bin
    ├── src
    ├── test
    └── MyFolder
        └── MyStatchert.ysc
```

Note that the second step, which is optional, has also been done and that the name chosen for the source folder that will contain the JUnit test class is *test*. At this point, the step 5 command should look like this:

```
java -jar .\CREATest-0.0.2.jar
    -scc C:\path\to\itemis_CREATE\scc.bat
    -projectPath C:\path\to\workspace\MyProject
    -sourceDir MyFolder -sourceFile MyStatechart
    -targetPackage statechart
    -evoTestDir test -evoSearchBudget 10
```

Note that the value of the optional `-evoTestDir` option corresponds to the source folder created in the second step. During the execution, some useful information is print to video, such as which phase of the process is being executed, warnings and errors. After running the JAR, the project should look like this:

```
C:\path\to\workspace
└── MyProject
    ├── bin
    │   └── ...
    ├── src
    │   ├── com
    │   │   └── yakindu
    │   │       └── core
    │   │           ├── IEventDriven.java
    │   │           └── IStatemachine.java
    │   └── statechart
    │       ├── MyStatechart.java
    │       └── MyStatechartSimplified.java
    ├── test
    │   └── statechart
    │       ├── MyStatechartSimplified_ESTest_scaffolding.java
    │       └── MyStatechartSimplified_ESTest.java
    ├── MyFolder
    │   ├── MyStatechart.ysc
    │   ├── MyStatechart.sgen
    │   └── MyStatechartSimplifiedTest.sctunit
    └── evosuite-report
        └── statistics.csv
```

Note that which classes are generated in the com.yakindu.core package depends on the input statechart and could be more (and even different) than those generated in this example. The most important artifacts generated by the process, that should be analyzed by the user, are: the MyStatechart.java file, which contains the Java implementation of the input statechart, the MyStatechartSimplified_ESTest.java file, which contains the JUnit test cases generated by EvoSuite, and the MyStatechartSimplifiedTest.sctunit file, which is the actual output of the whole process and contains the SCTUnit test cases that can be executed on the input statechart.

Acknowledgements

First, I would like to express my deepest appreciation to my advisor, Prof. Angelo Gargantini, and my co-advisor, Dr. Andrea Bombarda, for guiding me with patience and constant support.

Thanks should also go to the staff of itemis for providing me with help using their tool, itemis CREATE.

I would like to thank Michele Beretta for providing the LaTeX template [4] used during the writing of this thesis.

Lastly, I would like to extend my sincere thanks to my family, especially my parents and my brother. Their unconditional support helped me embark on this journey with serenity.