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

A Method for Testing and Validating Executable Statechart Models

Tom Mens, Alexandre Decan, Nikolaos I. Spanoudakis

Service de Génie Logiciel, Université de Mons, Belgium

Received: date / Revised version: date

Abstract Statecharts constitute an executable language for modelling event-based reactive systems. The essential complexity of statechart models solicits the need for advanced model testing and validation techniques. In this article we propose a method aimed at enhancing statechart design with a range of techniques that have proven their usefulness to increase the quality and reliability of source code. The method is accompanied by a process that flexibly accommodates testing and validation techniques such as test-driven development, behaviour-driven development, design by contract, and property statecharts that check for violations of behavioural properties during statechart execution. The method is supported by the *Sismic* tool, an open source statechart interpreter library in Python, that supports all the aforementioned techniques. Based on this tooling, we carry out a controlled user study to evaluate the feasibility, usefulness and adequacy of the proposed techniques for statechart testing and validation.

1 Introduction

Statecharts were introduced nearly three decades ago by David Harel [29, 30] as a visual executable modelling language. From a formal point of view, they can be considered as an extension of hierarchical finite state machines with characteristics of both Mealy and Moore automata. Statecharts (sometimes referred to as *behavioural state machines*) are part of the UML standard [2] and constitute a popular modelling notation for representing the executable behaviour of complex reactive event-based systems. They are frequently used in industry for the development of real-time systems and

Send offprint requests to:

embedded systems [15,44], relying on commercial tools such as IBM Rational's *StateMate* and *Rhapsody*, the Mathworks' *Stateflow*, itemis' *Yakindu Statechart Tools*, and IAR Systems' *visualSTATE*. There are also multiple open source frameworks defining domain specific languages (DSLs) based on statecharts, such as the ATOMPM [47] web-based domain-specific modeling environment with support for model transformation, and the AMOLA [46] language targeting the multi-agent systems community. Most of these tools support visualisation, modification and simulation of statecharts, as well as code generation from statechart models. The more advanced tools also provide support for model debugging and model verification.

At the level of source code development, a variety of testing techniques have found widespread use. These include *test-driven development (TDD)* [5], *behaviour-driven development (BDD)* [41], and *design by contract (DbC)* [39]. Most of these techniques have not been taken up at design level, however, for increasing the quality and reliability of behavioural software models. In particular, it is poorly understood how such techniques can be used for testing and validating executable statechart models. Indeed, designing statecharts and their interaction with the environment can be quite complex and error-prone, partly because of the statechart formalism itself, and partly because of the complex behaviour that these statecharts are modeling [13].

To address this problem, we propose a *method* aimed to enhance statechart design with such techniques. The method comes with dedicated tooling, in the form of *Sismic* (a recursive acronym for "Sismic Interactive Statechart Model Interpreter and Checker"), a modular Python library composed of a statechart interpreter and associated libraries for testing and validation of executable statecharts based on the techniques of TDD, BDD, DbC, and property statecharts that allow to monitor for violations of behavioural properties during statechart execution. The library is provided as an open source solution, in order to facilitate its extension by other researchers. It comes with support for importing and exporting statechart models to and from external statechart visualisation and editing tools.

The remainder of this article is structured as follows. Section 2 explains and motivates the techniques of TDD, BDD, DbC and runtime monitoring. Section 3 presents the proposed method and process for statechart design and testing. Section 4 explains the tooling that we have developed to support all the techniques supported by the method. Section 5 and Section 6 illustrate how to use the method and its associated tools by means of an example. Section 7 evaluates the method and tools through a controlled user study. Section 8 compares our work with other approaches, and Section 9 concludes.

2 Background

2.1 TDD and BDD

Test-Driven Development (TDD) [5] has become accepted for source code development thanks to the many available frameworks for automated *unit testing*. TDD is often accompanied by so-called *user stories*, representing informal, natural language descriptions of one or more features of the software system, written from an end user perspective.

The technique of *Behaviour-Driven Development (BDD)* allows to bridge the gap between user stories and executable functional tests. BDD extends TDD with acceptance test or customer test-driven development practices as found in extreme programming [41,50]. BDD allows users to specify representative *scenarios* and their expected outcomes using *Gherkin*[1], a domain-specific natural language that has been created specifically to support BDD. It enables users to describe the intended software behavior without needing to detail how that behavior is or will be implemented. Scenarios and their outcomes are described as sequences of *steps* expressed as domain-specific natural language sentences using *Gherkin*-specific keywords such as `Given`, `When`, `Then`, `And` and `But`. Developers can then define mapping code for these steps, in order to enable the automatic execution of each of these scenarios as executable functional tests. As such, the technical gap between developers and users is reduced.

User stories and BDD would provide a very good basis for coming up with relevant functional tests for executable statecharts, assuming that developers are available to write the necessary mapping code to execute the scenarios as functional tests over the statechart.

2.2 DbC

Design by Contract (DbC) was introduced more than two decades ago by Bertrand Meyer and popularised through the object-oriented programming language Eiffel [39,21]. It is often used to support the interaction and composition of software components (e.g., methods, functions, classes or packages) in complex systems based on the interface specifications of these components. DbC prescribes that software components should have formal, precise and verifiable interface specifications, which extend the ordinary definition of abstract data types with preconditions, postconditions and invariants. The term *contract* stresses the obligation of the programmer to respect the conditions of the code she is using.

The use of DbC has been shown to increase the reliability of executable code. It has been used with success to enable automated debugging with AutoFix, a tool that uses contracts to fix faults in general-purpose software [43]. Other programming languages also provide support for DbC. For Java

[1] See `http://docs.behat.org/en/v2.5/guides/1.gherkin.html`

programs for example, DbC is supported by JML, a formal behavioural interface specification language [36].

OCL, the constraint language that is part of the UML standard, offers the possibility to specify preconditions, postconditions and invariants on UML models [1]. It has been used to specify contracts on class diagrams [8] supported by the USE tool [25], and has recently been extended to support behavioural models such as protocol state machines [28] and sequence diagrams [26]. We are not aware, however, of any DbC tool that provides integrated support for specifying and checking structural properties during the runtime execution of statecharts.

Just like contracts can be defined on source code components at different levels of granularity, it is useful and desirable to express contracts at different levels on an executable statechart. Contracts could be defined on individual states (either basic or composite states) to express how these states are supposed to interact with other states. Contracts could be expressed on concurrent regions of a state to specify how these regions are supposed to synchronise with each other. Finally, contracts could be specified on transitions to verify that any action executed by a transition satisfies some invariants. Once defined, contracts can be monitored continuously while the statechart is being executed.

2.3 Runtime Verification

Any mechanism for monitoring observable behaviours or properties over an executing system is considered *runtime verification* [37]. It is also known under the names *runtime monitoring* or *dynamic analysis*. The technique is considered *lightweight*, avoiding the complexity of formal verification techniques by working directly with the actual system (as opposed to a more abstract formal representation of the system) and analyzing only a limited number of execution traces. It can be regarded as a testing approach, since it only checks for violations of properties at runtime, but cannot make any guarantees that the properties will hold for all possible system runs. Techniques and formalisms to monitor system properties during its execution include regular expressions, temporal logics, state machines and rule-based programming [23].

In the context of our work, the system being monitored will be an executable statechart. Drusinsky proposed TLCharts, an extension of statecharts allowing to specify Temporal Logic assertions and monitor violations of these assertions at runtime [16,17]. While the usefulness of such logic formalisms seems without doubt, their usability has been criticised, and different attempts have been made to increase the usability by non-logicians [18,11,6]. We will therefore explore an alternative approach, using the full expressive power of statecharts to monitor properties over statecharts.

3 Process

The main contribution of this work is to provide a process with associated tool support for the phases of statechart design and statechart testing. Statechart testing enables to validate the design before its integration during the actual system implementation.

We define our *process* as a *method fragment* [12,33] to be used by the designer in her specific software or systems development projects involving executable statechart modeling. Method fragments act as reusable methodological parts that can be flexibly used as puzzle pieces, allowing to accommodate the process to the specificities of any project, in function of the current needs and available competencies. Since the method fragment we propose will focus on the design phase only, we assume that a preceding analysis phase has already been carried out, and that the statechart design (and associated testing) phase will be typically followed by an implementation and deployment phase.

Our method fragment is defined using the SPEM 2.0 language for representing software methodologies [42] as a series of phases containing *tasks* that output *work products* that are required as input for tasks in later phases. Fig. 1 provides a high-level view of the different involved phases. Fig. 2 shows the different work products created, used and modified throughout these phases.

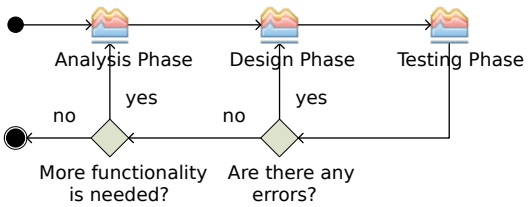


Fig. 1 SPEM 2.0 activity diagram describing the proposed process.

The method fragment assumes that in the overall software methodology there is an *Analysis Phase* that outputs a *component diagram* defining the different components of the system, and *user stories* expressing how the system should behave. For systems that require user interaction we also assume that a *UI mockup* is created during the analysis phase.

Using these work products as input, the statechart *Design Phase* will commence. The key work product created during the design phase is the *statechart*, a pivotal work product for any statechart-based system design. The *Design Phase* is followed by a *Testing Phase*, providing different types of *test results* as output. If any errors or failures are reported in the test results, the process iterates over the design phase to address these errors. If all errors have been fixed, a new development iteration can start from the analysis phase to implement more functionality, if needed. At this point,

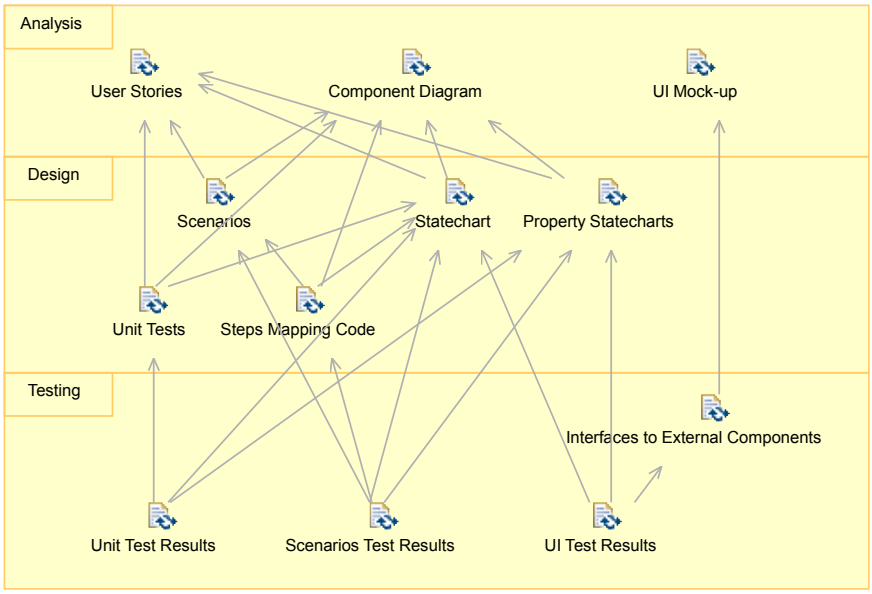


Fig. 2 SPEM 2.0 work products dependencies diagram.

the development team can decide to hook an implementation or integration phase to continue to develop the product – knowing that the designed statechart has been validated.

In the next subsections we detail the statechart design and testing phase in terms of their specific tasks and work products.

3.1 Design Phase

Fig. 3 shows all tasks of the statechart Design Phase. Tasks at the same horizontal level can take place concurrently. Fig. 4 complements this view by providing for each task, its input and output work products along with the role achieving the task.

The following tasks are part of the design phase:

- *Define Statechart* is undertaken by a *Software Engineer*. He reads the *user stories* (defining the behavior of the system) and the *component diagram* and produces the *statechart* that captures this behavior. This task is pivotal in the design phase as it unlocks many other tasks.
- In *Enrich Statechart with Contracts*, the *Software Engineer* follows a DbC approach to augment the *statechart* with contracts composed of preconditions, postconditions and invariants over states and transitions.
- *Define Scenarios* is carried out by a *Tester*. He is in charge of converting the free text *user stories* into BDD *scenarios*.
- In *Implement Scenario Steps*, a *Programmer* needs to write the *mapping code* from the *steps* used in scenarios to statechart test primitives. This

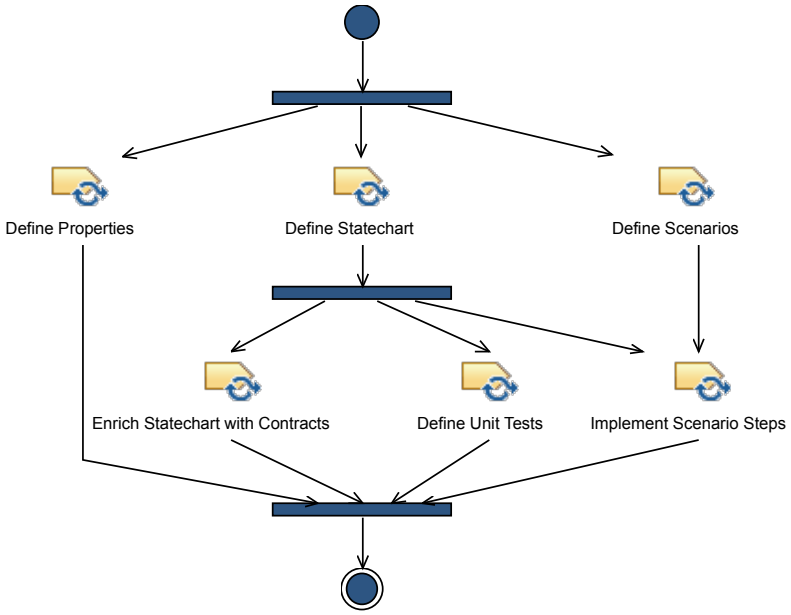


Fig. 3 SPEM activity diagram describing the statechart Design Phase. Black rectangles indicate fork or join nodes. Figure created using the Eclipse Process Framework (EPF) Composer, a tool for producing customizable software processes.

code will be used during the testing phase to check the scenarios over the statechart.

- *Define Properties* is concerned with defining behavioural properties over the system. It can be desirable to express functional properties of the intended behaviour, for example in terms of the events that are consumed and sent by a particular component. Such properties can be verified dynamically during the execution of the component. The *Software Engineer* relies on the *component* diagram along with the *requirements* and her domain knowledge to define these properties.
- *Define Unit Tests* involves writing *unit tests* for testing the *statechart* in a later phase. It is carried out by a *Programmer*, since it requires knowledge of the programming language used by the unit testing framework.

3.2 Testing Phase

The statechart Testing Phase relies on the output work products of the Design Phase to test and validate statecharts using a range of different techniques: dynamic monitoring of contracts and properties, running unit tests and scenarios over the statechart. Fig. 5 presents the tasks of this phase and the details of each task in terms of roles and work products:

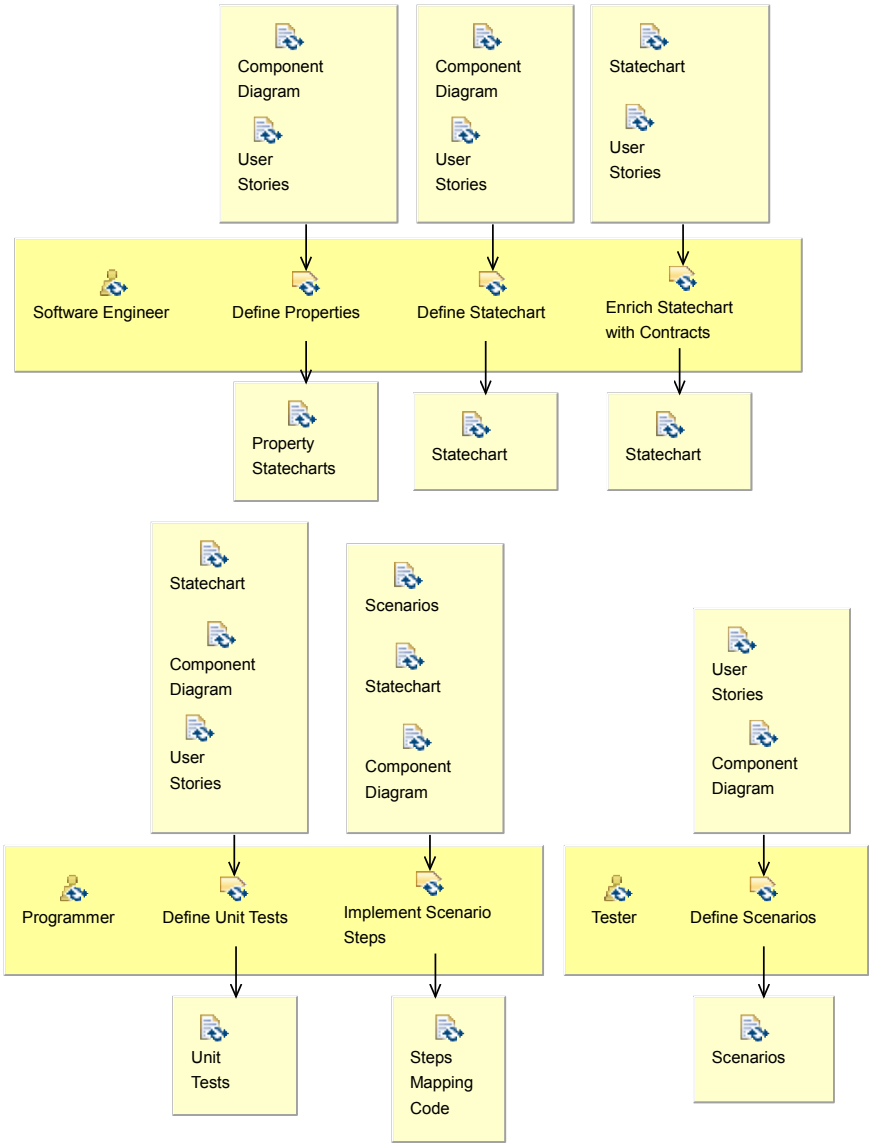


Fig. 4 SPEM 2.0 activity detail diagram for the design phase. Every task is connected to input and output work products along with the role of the person executing the task.

– The task *Integrate External Components to Statechart* allows the *Programmer* to realise *interfaces to external components* that need to participate in the testing process. If there are no external systems or user interfaces available this task is not needed. Hence, this task can be used for testing just the validity of the statechart in isolation (by simulating

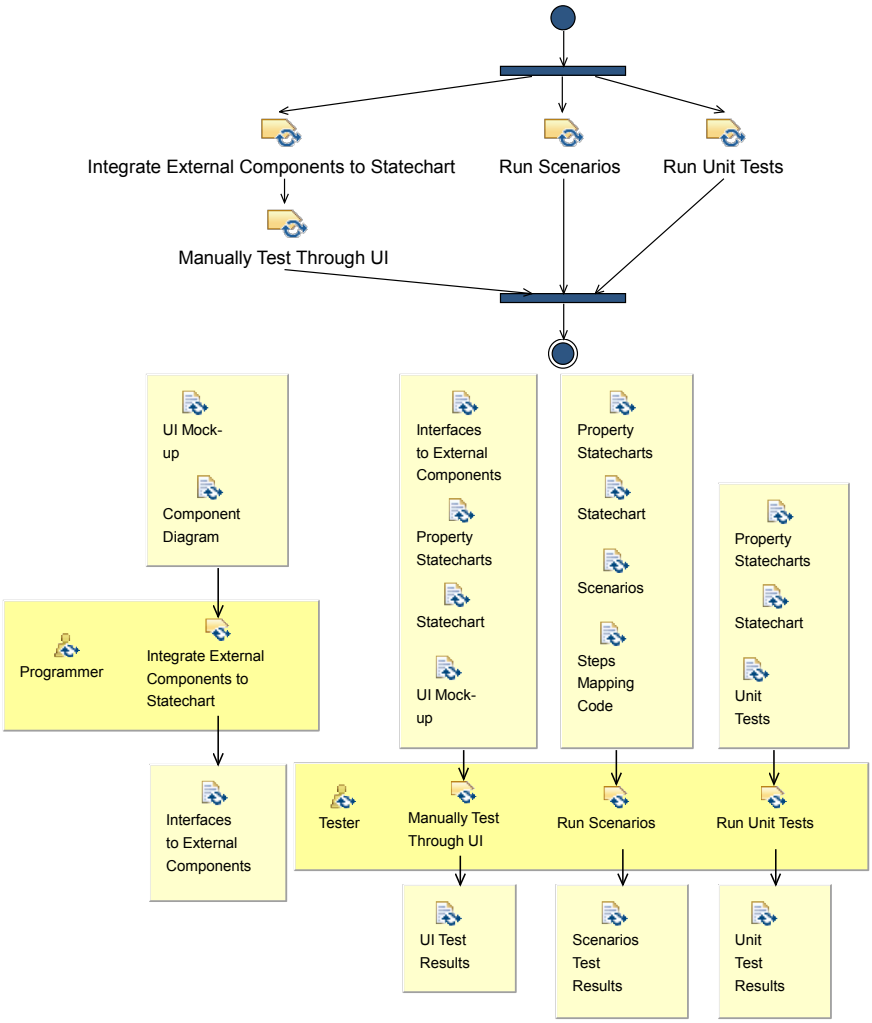


Fig. 5 SPEM activity diagrams for the statechart testing phase.

the external system events), or during a later implementation phase with all external systems connected.

– *Manually test through UI* can be done by a *Tester* to manually explore and validate the functionality of the statechart by means of an external user interface that has been created for this purpose, and connected to the statechart.
– *Run Scenarios* can be done by the *Tester*, by executing the scenarios (by means of the provided steps mapping code) with a BDD tool and analysing the scenarios test results. In more complex cases, this task may involve selecting which scenarios to execute.

– *Run Unit Tests* can be done by the *Tester*, by providing the already created unit tests to the test runner of a unit testing framework and analysing the unit test results. In more complex cases, this task may involve selecting which tests to execute.

It is important to stress that, during the three tasks carried out by the *Tester*, all statechart contracts and properties that were specified as part of the design phase, will be dynamically monitored for violations. Hence, the *test results* will also include reports of any such violations.

4 Tooling

The process presented in Section 3 needs to be supported by automated tools in order to be of any practical use. Such tooling is required, because designing and testing statecharts can be quite hard due to their complexity and because there are many subtleties of the statechart formalism. Hence, it is difficult to formally guarantee conformance of a statechart to its intended requirements. Our method proposes different ways to test and validate statecharts, by using the techniques of unit testing, BDD, and dynamic monitoring of contracts and properties expressed over statecharts.

To automate and support the proposed method we developed the *Sismic* tool. *Sismic* is a Python library for interpreting and testing statecharts. Version `0.26` has been used for this article. It is distributed through the Python Package Index (`pypi.python.org/pypi/sismic`). Its documentation can be found on `sismic.readthedocs.io`. Its source code is available on `github.com/AlexandreDecan/sismic` under the open source licence LGPLv3, adhering to the principles of open science and open research. It allows other researchers to use and extend the tool, and it facilitates integrating received feedback into newer versions of the tool.

The high-level architecture of *Sismic* is summarised in Fig. 6. The different code components and work products of this architecture will be presented in the following subsections. All parts of the architecture included in the shaded gray box labeled **sismic** have been developed specifically for the purpose of statechart simulation, testing and validation. For some of these activities, *Sismic* makes use of third-party libraries (e.g. `behave` for BDD) or external tools (e.g. PlantUML and ASEME for model visualisation and editing). Input and output files used by *Sismic* are shown in yellow note boxes.

4.1 Importing and exporting statechart models

Internally, *Sismic* encodes statechart models as objects. Experienced Python developers may choose to directly create and manipulate statecharts in this way through the model API that has been provided for this purpose.

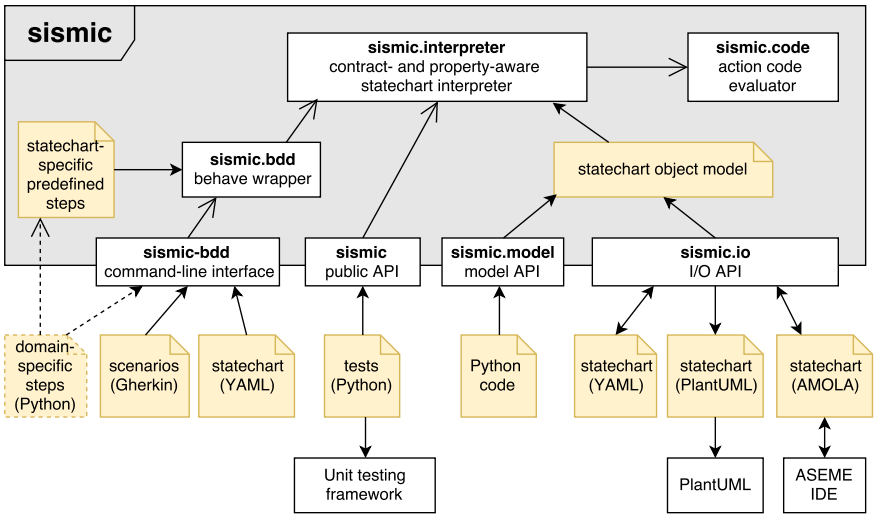


Fig. 6 Architectural overview of the *Sismic* tool framework

In practice however, it is much more comfortable to create statechart models using either a text-based markup editor or an external visual editor, and import these models through the I/O API provided for this purpose. Upon import of a statechart file, the syntactic correctness of the statechart is checked automatically.

Currently, *Sismic* allows importing and exporting statechart expressed using the human-readable YAML markup language, and has experimental support for importing and exporting statechart diagrams expressed in the AMOLA language [46] through the ASEME IDE[2] statechart editor [48]. *Sismic* also provides export support to PlantUML[3] in order to visualise statecharts, benefiting from its automatic layout features. This is how we generated Fig. 13 for example. Other exchange formats can be easily accommodated.

4.2 Statechart interpreter

The core of *Sismic* is composed of its *statechart interpreter*. In order to execute a statechart model, a statechart interpreter must be instantiated. This interpreter relies on an *action code evaluator* to execute any action code contained in the statechart specification. By default, action code and guards (conditions) are expressed using regular Python code. Other action languages can be supported if a language interpreter is provided for them.

The statechart interpreter offers a discrete, step-by-step, and fully observable simulation engine, supporting the UML 2 statechart concepts and

[2] aseme.tuc.gr
[3] plantuml.com

semantics. By default, the statechart interpreter uses an inner-first/source-state and run-to-completion semantics (a.k.a. macro-step, super-step or big step semantics [32,31,20]). Since the UML specification [2] deliberately leaves the order in which transitions are to be executed undefined we needed to make some implementation choices. The interpreter processes eventless transitions before transitions containing events, and consumes internal events before external ones. However, *Sismic* also provides modular support for providing other semantics.

Parametrized events are supported as well. Event parameters can be accessed by the statechart through the attributes of the `event` variable. For example, if an event `heating_set` has a parameter `power`, its value can be accessed using `event.power` in action code, guard and contracts of any transition triggered by `heating_set`.

When simulating statecharts, simulated time (logical time) and wall-clock time are supported. By default, the statechart interpreter uses simulated time. Several time-related predicates (e.g., `after`, `idle`) can be used in guards, actions and contracts of statecharts.

4.3 Using the API to execute statecharts

Sismic's statechart interpreter can be used to control statecharts programmatically. The code fragment of Fig. 7 provides an example of how to do this, on the basis of a microwave controller statechart that will be used as running example in Section 5 and Section 6 (see Fig. 13).

```
1  from sismic.io import import_from_yaml
2  from sismic.interpreter import Interpreter
3
4  statechart = import_from_yaml(filepath='microwave.yaml')
5  interpreter = Interpreter(statechart)
6
7  interpreter.execute_once()
8  interpreter.queue('door_opened')
9  interpreter.execute_once()
10 interpreter.time += 2
11 interpreter.execute()
12
13 print('Active states:', interpreter.configuration)
14 print('Timer value:', interpreter.context['timer'])
15 print('Power value:', interpreter.context['power'])
```

Fig. 7 Code fragment of a statechart simulation with *Sismic*.

After instantiating a statechart interpreter with the statechart specification loaded from a YAML file (line 4), the statechart is put in its initial configuration by calling the `execute_once()` method (line 7). The `queue(...)` method stores a new `door_opened` event in the interpreter's event queue (line 8). Invoking the `execute_once()` method again (line 9) processes the event and returns a `MacroStep` instance. As illustrated in Fig. 8, if a macro

step processes a triggered transition, it includes every consecutive stabilisation step (including all steps needed to enter nested states, or to enter the configuration of a history state). Line 10 increases the simulated time by modifying the interpreter's time variable. Line 11 executes the statechart as long as a `MacroStep` is returned by the interpreter. Line 13 displays the currently active states of the statechart, and lines 14 and 15 display the values of the statechart's local variables timer and power.

4.4 Runtime monitoring of statecharts

A key feature of *Sismic*'s statechart interpreter is its built-in support for monitoring for violation of contracts and undesirable runtime properties. To this extent, the traditional macro- and micro-step semantics of statechart execution is *augmented* with some additional steps. This is illustrated in Fig. 8, which depicts (a deliberately simplified version of) the semantics of the statechart interpreter. For ease of readability we use the statechart notation to illustrate all steps that are followed by the interpreter when a statechart receives an event that triggers a transition. The "normal" semantics of statechart execution is shown by the yellow states.

The red states (with white font) in Fig. 8 augment the statechart semantics with runtime support for contract monitoring. They indicate where and when the contract checking (of states and transitions) intervenes to support DbC. This depends on whether it concerns pre- and postconditions or invariants. State preconditions are checked *before* the state is entered (i.e., before executing its entry actions, if present), state postconditions are checked after the state is exited (i.e., after executing its exit actions, if present), and state invariants of each active state are checked at the end of each executed macro step (corresponding to a stable running configuration of the statechart). Transition preconditions are checked *before* processing the transition (and before executing its optional transition action), transition postconditions are checked *after* processing the transition (and after executing the optional transition action), and transition invariants are checked twice: before and after processing the transition. Hence, transition invariants can be considered as syntactic sugar for conditions that are both preconditions and postconditions.

The statechart interpreter also provides support for monitoring properties at runtime. To avoid a statechart designer needing to learn a different (formal) language for expressing such properties, these properties can be expressed using the full expressive power of the statechart notation itself. Such properties are therefore referred to as *property statecharts*. Examples of such property statecharts are provided in Section 5.2.

Since these properties need to monitor a running statechart, their behaviour has to be expressed in terms of the events that are consumed or sent, or in terms of the states that are entered or exited by the statechart being monitored. To this extent, the statechart interpreter will raise specific

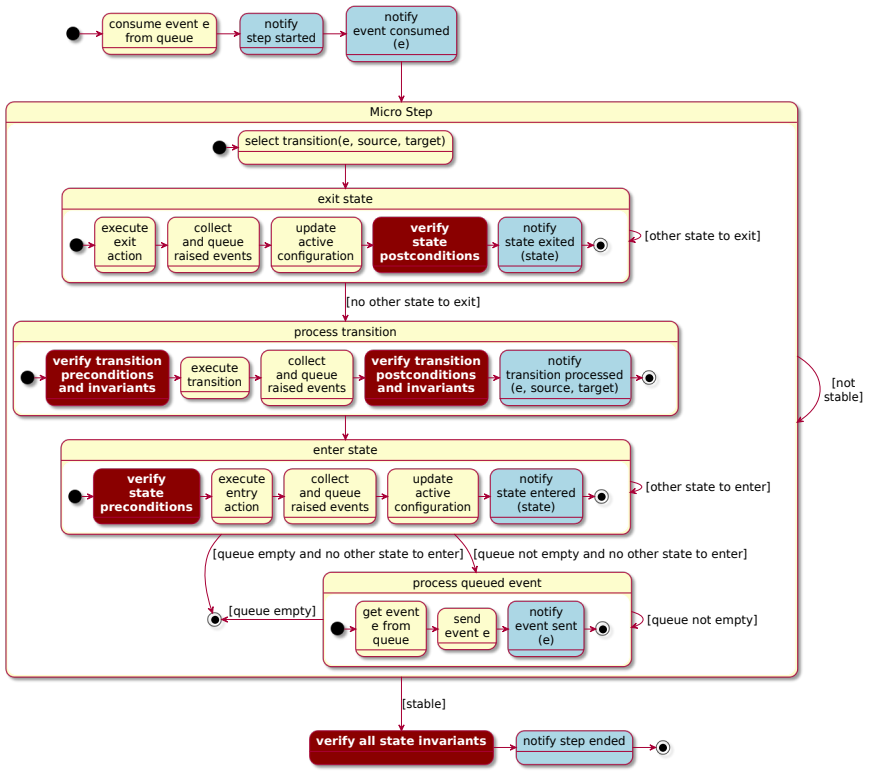


Fig. 8 *Sismic*'s macro-step semantics, adding support required for dynamic monitoring of contracts and properties.

meta-events about the execution of the statechart. The blue states in Fig. 8 (e.g., notify event consumed) show which meta-events are created during statechart execution. Some meta-events exhibit additional parameters: for example, when a state is entered, meta-event state entered has a parameter `state` whose value is the name of the state being entered. Similarly, meta-event event consumed has a parameter `event` that corresponds to the event being consumed.

Meta-events are automatically sent during statechart execution to all bound property statecharts. To bind a property statechart, it suffices to provide this property statechart to the `bind_property_statechart` method of an interpreter. These property statecharts monitor for property violations based on those meta-events, following a "fail fast" approach: a failure will be reported as soon as the monitored behaviour leads to a final state in one of the bound property statecharts.

Due to the meta-events being considered and the "fail-fast" approach adopted by the statechart interpreter for their verification, property statecharts are mainly intended to check for the presence of undesirable behavior (safety properties), i.e., properties that can be checked on a (finite) prefix

of a (possibly infinite) execution trace. While it is technically possible to use property statecharts to express liveness properties (something desirable *eventually* happens), this would require additional code for their verification since liveness properties are not supported "as is" by *Sismic*.

4.5 Support for unit testing and BDD scenarios

As for any other library, unit tests can be written for *Sismic* using its API to manipulate the statechart and hence, the unit tests themselves have to be expressed in the language supported by that framework (in our case, Python).

To evaluate BDD scenarios, *Sismic* provides a command-line interface (CLI) called `sismic-bdd`, which takes as input feature files containing scenarios expressed in the Gherkin language.

5 Running Example - Statechart Design Phase

To illustrate how *Sismic* supports the *Design Phase* of the method proposed in Section 3.1 we use a microwave oven controller as running example. The example is inspired by Gomaa [27]. Subsection 5.1 presents the work products (resulting from the Analysis Phase) that will be used as input for the Design Phase. Each of the remaining subsections correspond to one of the tasks of the Design Phase.

5.1 Analysis Phase Work Products

Component diagram. We assume that a microwave oven contains different interacting components controlled by a main Controller component through event-based communication (Fig. 9). The Controller receives events from the Door to signal when it is opened or closed, and from a WeightSensor to detect whether a food item is placed in, or removed from the oven. The User Input component represents the user interface. It contains three subcomponents: a Cooking component providing buttons that trigger events to start or stop cooking, and a Power and Timer component allowing to increment, decrement or reset the heating power and cooking time, respectively. The Controller uses two integer variables power and timer to keep track of the desired heating power and remaining cooking time. It uses a Display component to inform the user, by displaying a character string on the screen. A Clock component sends a tick event to the Controller every second. The Controller is able to switch on or off an indicator light controlled by the Lamp component. It also controls a Heating component (the magnetron device emitting the microwaves) by setting its power, and turning it on or off. It instructs the Turntable component to start or stop turning. A Beeper component can be used to make sound signals by sending it a beep event.

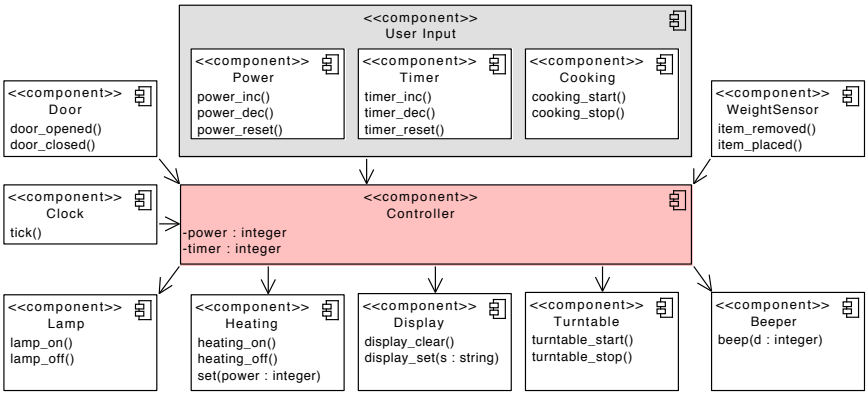


Fig. 9 Component diagram for a microwave oven. Each component lists the events it is able to send to a connected component. Some events are parameterised (e.g., display_set(s:string) and beep(d:integer)). The arrows on the component connectors indicate in which direction events are sent. Internal variables used only by the Controller component are preceded by a - sign.

UI mockup. Fig. 10 presents the design of a very simple user interface mock-up, illustrating how the user is supposed to manually control the microwave oven's components. The panel at the right represents icons and buttons for the Display, Beeper and User Input components, as well as a button to simulate the opening of the Door. The left panel shows the status of the Lamp, Turntable, WeightSensor and Door components. It also contains a button to close the Door.

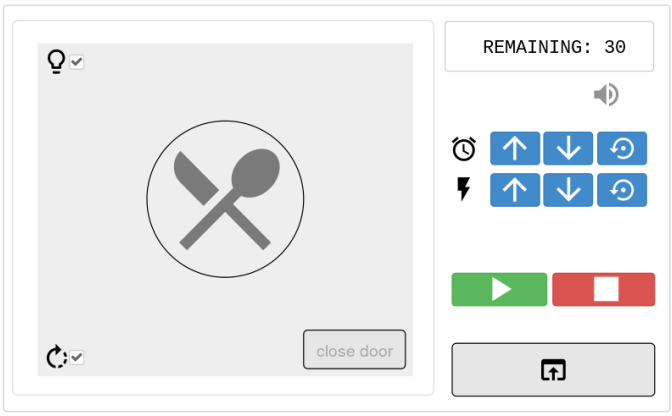


Fig. 10 Simple user interface mockup for a microwave oven simulation.

User stories. Any device that is supposed to interact with a user should have a set of functional requirements representing the intended functionality from the user point of view. This functionality can be expressed informally in terms of *user stories* expressing the intended outcome of typical interaction of the user with the device. These semi-structured textual user stories are provided by a domain expert who does not necessarily have experience with software modelling. A partial example of such a user story for the microwave oven is presented in Fig. 11.

> *"As a user, I want to be able to open and close the oven door to place my food in the oven. I want to be able to adjust the cooking time and heating power by pressing buttons. I want to use buttons to start and stop the cooking in order to control when the cooking takes place. I want the oven lamp to be on while the door is open so that I can see where to put my food. I also want the lamp to be on during cooking so that I can monitor the heating proces. Because I do not want to be exposed to dangerous microwaves, I don't want any microwaves to be emitted while the door is open."*

Fig. 11 Example of a user story for a microwave oven.

5.2 Define Properties

Considering the user story of Fig. 11, it is the task of a software engineer to express and enforce the safety criterion that microwaves should not be emitted while the oven door is open. It is the Heating component that is in charge of emitting microwaves. The Controller component implicitly assumes that, whenever a heating_on event is sent, the microwaves start emitting, and whenever a heating_off event is sent, the microwaves stop emitting.

The software engineer can express *property statecharts* that monitor the execution of the Controller component for violations of the safety criterion. The two property statecharts of Fig. 12 encode this expected behaviour. Their visual representation is automatically generated by the tooling presented in Section **??**. The first property statechart monitors if the oven stops emitting microwaves while the door is opened. This is ensured by checking that the heating_off event is sent by the Controller sufficiently rapidly (i.e., before the next tick is consumed). Monitoring that heating_off event is sent by the controller can be done by checking that a meta-event event sent is received by the property statechart, and that its parameter `event` has an attribute `name` equal to "heating_off" (encoded by the guard `[event.event.name == 'heating_off']`). If a tick has been consumed before heating_off is sent by the Controller, the property statechart will go to its final state, indicating a violation of the property.

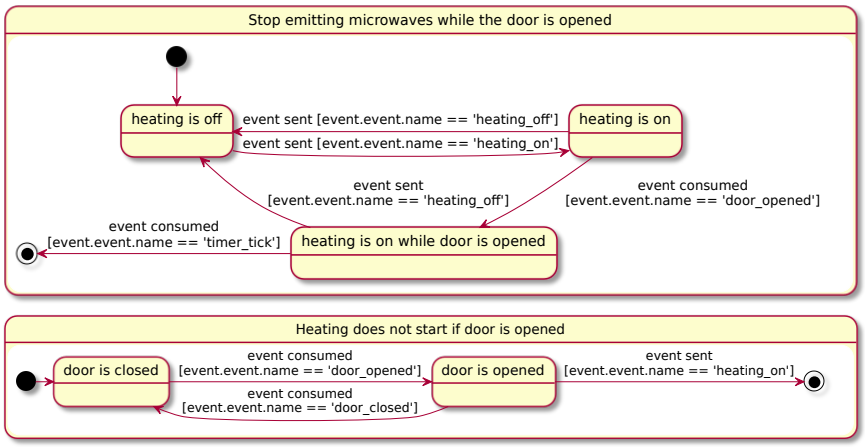


Fig. 12 Two property statecharts expressing the safety criterion that microwaves should not be emitted while the oven door is open. They monitor the Controller component for violations during its execution.

The second property statechart of Fig. 12 verifies that heating (i.e., emission of microwaves) does not happen while the door is still open. If the door is open and a heating_on event is sent before the door is closed again, the property statechart will go to its final state.

Fig. **??** illustrates another property statechart that is unrelated to the microwave's satefy criterion. Instead, it serves to check if heating is properly controlled by the microwave controller, by verifying that the events heating_on and heating_off are strictly alternating.

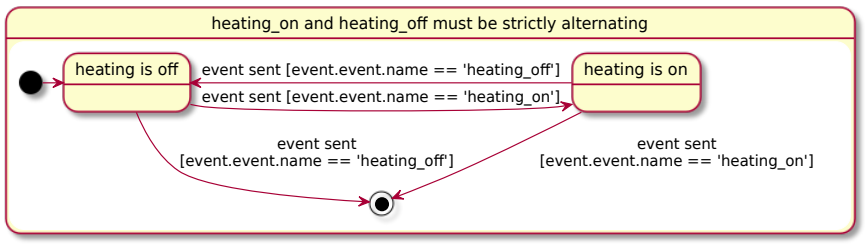


Fig. 13 Property statechart verifying the alternating behaviour of heating.

5.3 Define Statechart

It is the task of a software engineer to model executable statecharts whose event-based behaviour respects the functionality specified by the requirements, compatible with the events defined in the component diagram and the behaviour defined by the user stories.

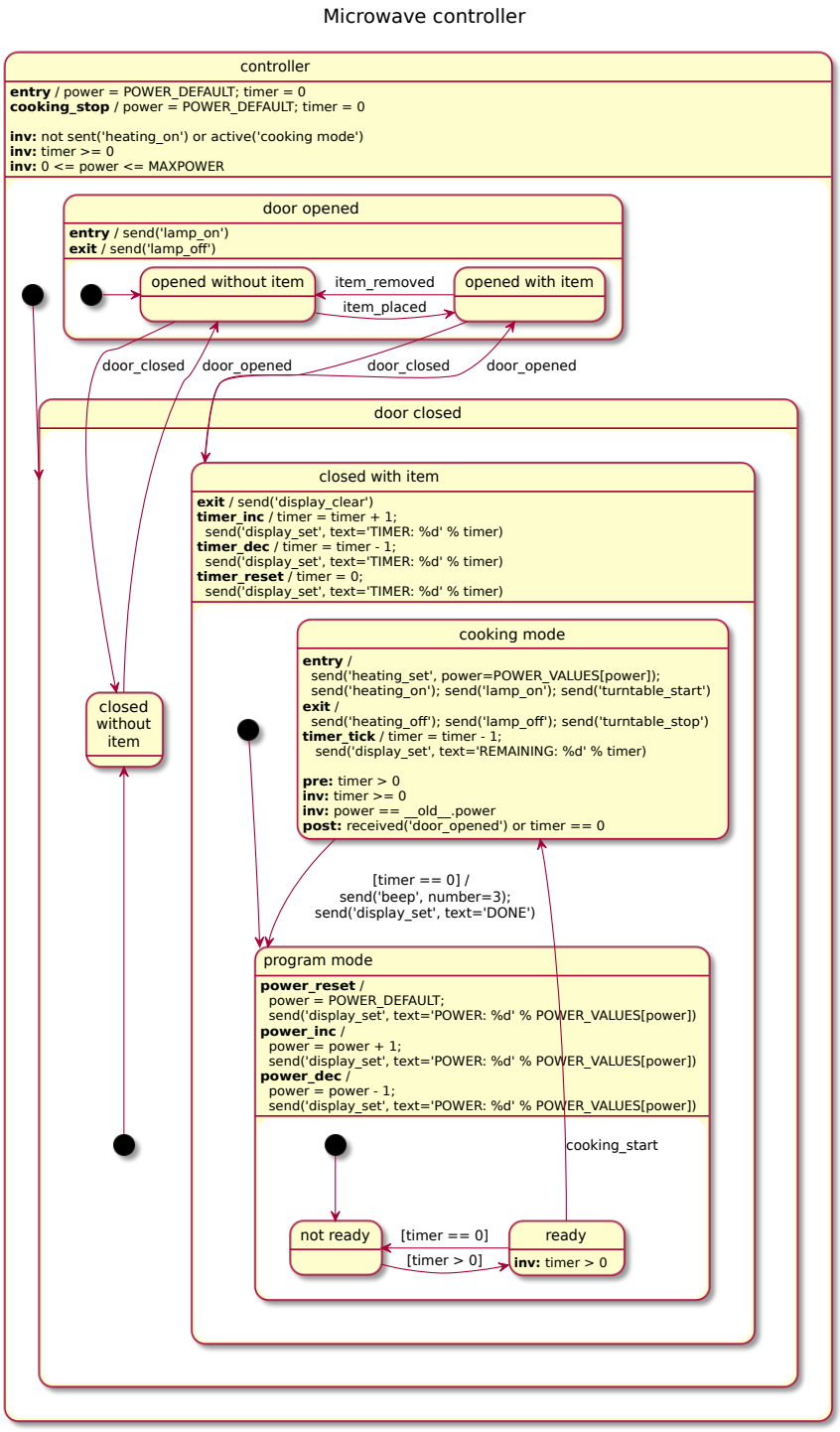


Fig. 14 Statechart modelling the executable behaviour of the Controller component.

Fig. 13 specifies the design of the Controller component as an executable statechart. The visual representation used in the figure is automatically generated by the tooling presented in Section 4. The used action language is Python. The statechart assumes the presence of three variables that represent the set of possible power values (expressed in Watt) and the default and maximum power of the microwave oven. They are initialised as follows:

```
POWER_VALUES = [300, 600, 900, 1200, 1500]
POWER_DEFAULT = 2  # 900 Watts
MAXPOWER = 3       # 1200 Watts
```

The statechart's behaviour depends on whether the door is opened or closed, as well as on whether an item has been placed in the oven for heating. When the door is closed with an item inside the oven, the user can set two internal variables of the statechart: the cooking timer (through the Timer component interface) and the requested heating power (in the program mode state, by using the Power component interface). The user will be informed of the timer and power values through the Display component. As soon as a value for the timer has been set, the Controller can start cooking in the cooking mode state (when event cooking_start gets triggered). Every second (i.e., every time a tick is received from the Clock component), the remaining time will be decreased until cooking is finished. Cooking can be cancelled by pushing the stop button through the Cooking component (triggering the cooking_stop event), which will reset the timer. This is achieved by the internal transition defined on the top-level controller state. Cooking can be paused by opening the door during cooking (door_opened event). After closing the door (door_closed event), cooking can be resumed by pushing the start button again (cooking_start event).

5.4 Enrich Statechart with Contracts

As motivated in Section 2, it is desirable to apply DbC to executable statecharts, by expressing and checking contracts on states and transitions. Fig. 13 provides some examples of useful contracts enriching the Controller statechart. They are defined on the controller state, the ready state and the cooking mode state respectively.

Keywords `pre:`, `post:` and `inv:` are used to represent preconditions, postconditions and invariants, respectively. The notation `__old__` is used to refer to the old value of a variable (e.g., `__old__.power`). Contracts can rely on a range of useful predicates. For example, `active(state)` is used to check if `state` is in the active configuration of the statechart, `sent(event)` verifies if a particular `event` has been sent during the current step, and `received(event)` verifies if a particular `event` has been received during the current step.

The contract on the root state controller imposes several invariants. The invariant `not sent('heating_on') or active('cooking mode')` asserts that heating_on events can only be sent while residing in the cooking mode

state. This provides an alternative way of partly monitoring the microwave safety criterion. The other invariants verify the accepted range of values for the statechart's local variables timer (that should not have negative values) and power (whose integer value should range between 0 and some fixed constant MAXPOWER).

The contract on the ready state expresses an invariant timer > 0. The contract on the cooking mode state has one precondition that cooking time should be strictly positive. It also has two invariants: cooking time cannot go below zero; and power value should not be changed while being in cooking mode. Finally, it has one postcondition that cooking can only be interrupted by opening the door or by reaching 0 seconds of remaining cooking time.

These contract specifications can be used to reveal conceptual errors in the statechart specification by monitoring the statechart's execution for contract violations.

5.5 Define Unit Tests

Unit tests provide a straightforward way of testing the intended statechart behaviour. Python developers can write unit tests for statecharts by relying on Python's built-in *unittest* library (or any other library providing support for unit testing), combined with *Sismic*'s API for executing statecharts. After loading and initialising the Controller statechart, and executing a sequence of events, one can check whether the statechart behaves as expected using the usual assertion methods offered by the unit testing framework (e.g., assertEqual, assertNotEqual, assertNotIn, ...). For example, one can test whether the statechart resides in a particular state, whether it has sent or received a particular event, and whether its internal variables contain a specific value.

Fig. 14 illustrates two simple unit tests for the Controller statechart. The first test of Fig. 14 (lines 12-19) partially verifies the oven's safety criterion of not emitting microwaves while the door is open. The second test (lines 21-25) verifies that the internal timer variable increases when timer_inc is received.

Running unit tests in an automated way allows designers to verify if the statechart respects the intended behaviour, as specified by the requirements. Such unit tests can use the full power of a programming language to express complex tests. However, it assumes statechart designers to be fluent in the programming language that is used to express the unit tests, and requires them to know *Sismic*'s API . This is not necessarily the case, since domain experts may not have a profound knowledge of the programming language details of how to specify and run unit tests. Hence, the need to use both a modelling language (for expressing statecharts) and a programming language (for writing unit tests) introduces an unnecessary technical gap. It goes against the main principles of software modelling, which aims at hiding the accidental complexity and technical details of the underlying programming language.

```
import unittest
from sismic.io import import_from_yaml
from sismic.interpreter import Interpreter

class MicrowaveTests(unittest.TestCase):
  def setUp(self):
    with open('microwave.yaml') as f:
      sc = import_from_yaml(f)
    self.oven = Interpreter(sc)
    self.oven.execute_once()

  def test_no_heating_when_door_is_not_closed(self):
    self.oven.queue('door_opened', 'item_placed', 'timer_inc')
    self.oven.execute()
    self.oven.queue('cooking_start')
    for step in iter(self.oven.execute_once, None):
      for event in step.sent_events:
        self.assertNotEqual(event.name, 'heating_on')
    self.assertNotIn('cooking_mode', self.oven.configuration)

  def test_increase_timer(self):
    self.oven.queue('door_opened', item_placed', 'door_closed')
    self.oven.queue(10 * ['timer_inc'])
    self.oven.execute()
    self.assertEqual(self.oven.context['timer'], 10)
```

Fig. 15 Example of Python unit tests for the Controller statechart.

5.6 Define Scenarios and Implement Scenario Steps

To avoid needing to code tests programmatically, the technique of *BDD* allows to bridge the gap between informal user stories (such as the one shown in Fig. 11) and executable scenarios expressed using the *Gherkin* language. This allows the domain expert to express scenarios in a domain-specific natural language, and only requires the knowledge of a few specific *Gherkin* keywords for their writing. Fig. 15 provides a concrete but partial example of such scenarios.

In order to be able to execute scenarios, a Python developer needs to write code defining the mapping from the actions and assertions expressed as natural language sentences in the scenarios (using specific keywords such as given, when, and or then) to Python code that manipulates the statechart. *Sismic* already provides a set of predefined statechart-specific steps that can be used in scenarios, such as "`Given I send event {name}`" or "`Then event {name} should be fired`".

While these predefined steps should be sufficient to manipulate the statechart, it is sometimes more intuitive to use domain-specific steps to write scenarios. For instance, the domain-specific step "`Given I open the door`" corresponds to the action of sending an event door_opened to the statechart. The mapping from this domain-specific step to the action of sending a door_opened event to the statechart could be defined using plain Python code that accesses *Sismic*'s interpreter, as illustrated by the following code fragment:

```
Feature: Cooking

  Scenario: Start and stop cooking
    Given I open the door
    And I place an item in the oven
    And I close the door
    And I press increase timer button 5 times
    And I press increase power button
    When I press start button
    Then heating turns on
    When I press stop button
    Then heating turns off

  [...]
```

```
Feature: Lighting

  Scenario: Lamp is on when door is open
    When I open the door
    Then lamp turns on
    When I close the door
    Then lamp turns off

  Scenario: Lamp is on while cooking
    Given I open the door
    And I place an item in the oven
    And I close the door
    And I press increase timer button 5 times
    When I press start button
    Then lamp turns on

  [...]
```

```
Feature: Safety Criterion

  Background: Put food and prepare for cooking
    Given I open the door
    And I place an item in the oven
    And I close the door
    And I press increase timer button 5 times

  Scenario: No heating when door is not closed
    Given I open the door
    When I press start button
    Then heating does not turn on

  Scenario: Opening door interrupts heating
    Given I press start button
    And 3 seconds elapsed
    When I open the door
    Then heating turns off

  [...]
```

Fig. 16 Scenarios (expressed in Gherkin language) describing *part of* the intended functionality of a microwave oven.

```
from behave import given

@given('I open the door')
def open_the_door(context):
  return context.interpreter.queue('door_opened')
```

Alternatively, this domain-specific step can be implemented more easily as an alias of predefined step "`Given I send event door_opened`". To do so, *Sismic*'s API provides two convenient helpers to map new steps to predefined ones, namely `map_action` and `map_assertion`. Using these helpers, one can easily implement the domain-specific steps of Fig. 15, as illustrated by the following code fragment:

```
from sismic.bdd import map_action, map_assertion

map_action('I open the door',
           'I send event door_opened')
map_action('I place an item in the oven',
           'I send event item_placed')
...
map_assertion('Heating turns on',
              'Event heating_on is fired')
map_assertion('Heating does not turn on',
              'Event heating_on is not fired')
```

6 Running Example - Statechart Testing Phase

This section revisits our running example to explain how the *Statechart Testing Phase* (presented in Section 3.2) can be carried out in a semi-automated way based on the *Sismic* tool presented in Section 4. Each of the following subsections correspond to one of the tasks of the *Statechart Testing Phase*, summarised in the use case diagram of Fig. 16.

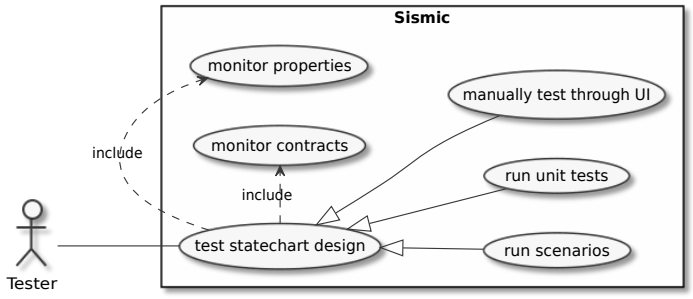


Fig. 17 Use case diagram summarising the tasks of the Statechart Testing Phase.

As shown in Fig. 16 there are three complementary ways of testing statecharts: through an external UI, by running unit tests, or by running scenarios. Independently of which of these techniques is being used (preferably

all of them), the statechart interpreter will also continuously monitor for violation of contracts and property statecharts.

6.1 Monitor contracts

Sismic's statechart interpreter supports run-time monitoring of contract violations. Unless if explicitly instructed to disable contract checking, the interpreter monitors all contract conditions during statechart execution. This implies that contracts are monitored by default when running unit tests or BDD scenarios. Contracts are specified directly as part of the statechart description (as illustrated in Fig. 13). Contracts can be specified using the action language supported by the action code evaluator (Python by default), and can use a range of useful predicates. If a contract is violated, a `PreconditionError`, `PostconditionError` or `InvariantError` is raised.

6.2 Monitor properties

Sismic's statechart interpreter provides built-in support for verifying *property statecharts* such as the ones of Fig. 12 and Fig. **??**. To monitor property statecharts at runtime, it suffices to bind them to the interpreter using its bind_property_statechart method. Property statecharts are automatically monitored by the sismic-bdd command-line interface if they are provided using the `--properties` parameter.

During the execution of the statechart, the interpreter will monitor for property violations by checking if the property statechart arrives in a final state. As soon as this happens, the interpreter raises a `PropertyStatechartError`. It is up to the statechart designer to decide how to cope with this violation (e.g., by executing appropriate exception handling code).

6.3 Run unit tests

Running unit tests that make use of *Sismic*'s API, such as those shown in Fig. 14, is straightforward. Developers can use their favorite unit testing framework to run the unit tests. For example, one could use Python's built-in *unittest* library in the usual way, as illustrated in Fig. 17.

```
$ python -m unittest tests_microwave.py

Ran 2 tests in 0.089s
OK
```

Fig. 18 Result of running the unit tests on the Controller statechart.

6.4 Manually test through UI

An intuitive way of validating the behaviour of a statechart design is by exploring its behaviour by means of a simply GUI (such as the one of Fig. 18) that directly interacts with its associated statechart. To achieve this, one needs to bind the statechart interpreter to the event handler of the GUI, and to instruct specific UI actions (such as pressing a button) to send events to the statechart interpreter or vice versa. The GUI shown in Fig. 18 has been implemented using Python's `tkinter` library. We refer the interested reader to *Sismic*'s online documentation for more details on how to integrate statecharts with external Python code.

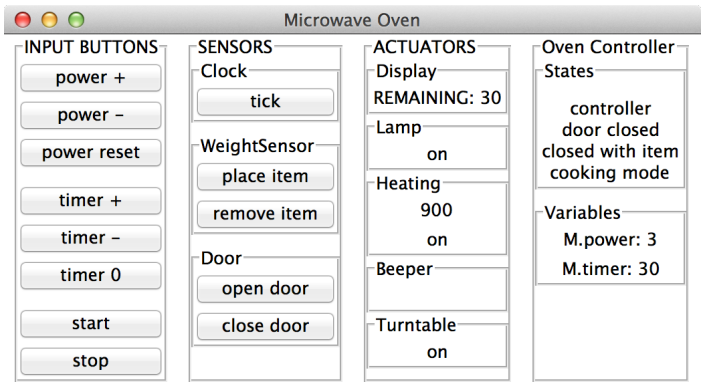


Fig. 19 GUI for interacting with the Controller statechart (implemented in Python with `tkinter`).

Exploring the executable statechart behaviour through this GUI will allow a tester to manually discover conceptual errors. For example, trying to decrease the timer too much (by means of the `timer -` button) will result in a violation of the invariant timer >= 0 on state controller because the action associated to event timer_dec in state closed with item decreases the value of timer (that was initialised to 0 by the entry action of controller) to −1. As a consequence, the interpreter raises an `InvariantError`, whose output is presented in Fig. 19.

This contract violation corresponds to a typical out of range error. It can be solved easily by adding a guard [timer > 0] to the event-action pair timer_dec / timer = timer - 1 defined on state closed with item in Fig. 13. The guard will prevent timer from being decreased if its value is already 0. In a similar way, guards should be added to the events power_inc and power_dec defined on state program mode to avoid violations of invariant 0 <= power <= MAXPOWER.

```
sismic.exceptions.InvariantError: InvariantError
Object: CompoundState('controller')
Assertion: timer >= 0
Configuration: ['controller', 'door closed', 'closed with item',
  'program mode', 'not ready']
Step: Step@0(
  Event('timer_dec'),
  [Transition('controller', None, event='timer_dec')], >[], <[])
Context:
 - POWER_VALUES = [300, 600, 900, 1200, 1500]
 - POWER_DEFAULT = 2
 - MAXPOWER = 3
 - power = 2
 - timer = -1
```

Fig. 20 Example output produced upon violation of a statechart contract.

6.5 Run scenarios

Assume that the scenarios of Fig. 15 are stored in .feature files, and that a mapping file steps.py has been defined containing the code required to map the natural language sentences of these scenarios to Python code that manipulates the statechart. The `sismic-bdd` command-line interface can then consecutively run each scenario contained in each feature file on the statechart (stored in file microwave.yaml). The result of the execution will be a summary of all executed scenarios and encountered errors (if any), as shown in Fig. 20.

```
$ sismic-bdd microwave.yaml
      --features cooking.feature lighting.feature satefy.feature
      --steps steps.py

Feature: Cooking

  Scenario: Start and stop cooking
    Given I open the door
    ...
    When I press stop button
      Assertion Failed: InvariantError
      Object: BasicState('cooking mode')
      Assertion: power == __old__.power
      ...
  ...

Failing scenarios:
  cooking.feature:3  Start and stop cooking

2 features passed, 1 failed, 0 skipped
5 scenarios passed, 1 failed, 0 skipped
28 steps passed, 1 failed, 1 skipped, 0 undefined
Took 0m0.130s
```

Fig. 21 Result of running the BDD scenarios on the Controller statechart.

The "*Start and stop cooking*" scenario (see Fig. 15) causes a violation of a contract during the execution of the step "*When I press stop button*".

Indeed, the cooking_stop event that is sent to the statechart when this step is executed triggers the internal transition of the root state controller. The execution of the action `power = POWER_DEFAULT` of the transition results in a modification of the value of the internal variable `power`, and thus in a violation of the invariant `power == __old__.power` specified for the cooking mode state.

It is up to the *Software Engineer* to decide to either forbid resetting the `power` variable, or to weaken the state invariant of cooking mode to allow changing the `power` variable when a cooking_stop event is received, e.g., `power == __old__.power or received('cooking_stop')`.

7 Evaluation

To evaluate the proposed method and the usefulness and usability of the proposed techniques for validating and testing statecharts, we conducted a controlled user study using the *Sismic* tool presented in Section 4.

7.1 Experimental Setup

The aim of the study was to evaluate:

- The usefulness of three of the proposed techniques for testing and validating statecharts, namely BDD, DbC, and dynamic monitoring for violations of property statecharts. We did not evaluate the unit testing technique, because we do not consider it as a new contribution and we did not expect all participants to have sufficient programming experience in Python unit test frameworks.
- The usability of the proposed implementation of the techniques in the *Sismic* tool.

Thirteen persons between 22 and 34 years old were selected to take part in the study based on convenience sampling. All participants were already familiar with the statechart formalism and all had a higher education degree (3 master students, 6 PhD students, and 4 postdoc researchers).

Each participant was handed out a questionnaire containing the instructions for the experiment. For reproducibility purposes, the full questionnaire and set of answers of each participant can be found on https://github.com/ecos-umons/sismic-validation.

After responding to a set of preliminary questions, participants were required to install *Sismic* on their machines. No particular installation problems were encountered. The actual experiment started from a simplified version of the microwave oven, for which the participants were first asked to understand and test its behaviour. Next they were asked to extend this functionality in a test-driven way, using the method, techniques and tools presented in this paper. To help in this activity, participants were provided with the following information at the outset of the experiment:

– A simplified version of the GUI of Fig. 18 to simulate the oven's behaviour.
– A textual description (YAML file) and visual rendering (similar to Fig. 21) of the statechart of the oven controller.
– An explanation of how the GUI interaction (e.g. button presses) has been mapped to statechart events.
– Executable BDD scenarios (expressed in Gherkin) for the oven's behaviour.
– Steps files (in Python) expressing the mapping of the scenarios to instructions for the statechart interpreter.
– An example of a property statechart.

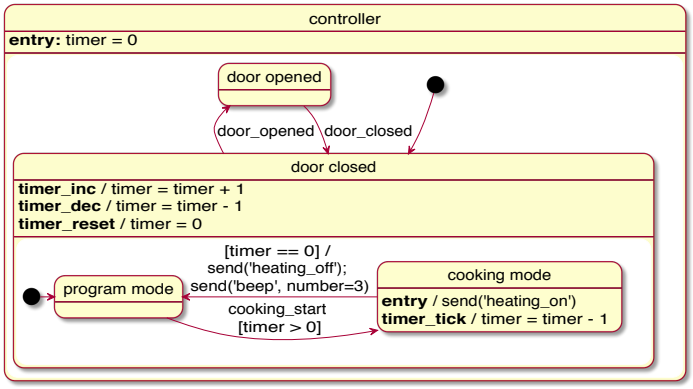


Fig. 22 Statechart of a simplified microwave oven controller.

The experiment was composed of 9 successive tasks (T1 to T9). As illustrated in the rightmost boxplot of Fig. 22, the total experiment took between 58 and 338 minutes (median value of 162), depending on the participant. With a few exceptions, the tasks increased in complexity, reflected by an increasing time required to carry them out.

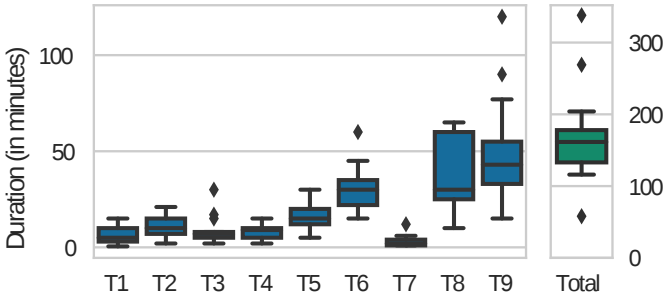


Fig. 23 Boxplots showing the duration of each task (blue) and in total (green).

Tasks T1 to T4 were devised to get acquainted with the provided example and files. During these tasks the participants were expected to play the role of a Tester who has not been involved in the statechart design but is required to test and validate the statechart. The tasks were the following:

T1 corresponds to task *Manually test through UI* of the Testing Phase of the proposed process. Participants were asked to get familiar with the provided GUI, and to use it to discover possible problems in the oven behaviour.

T2 is a preparatory task to make the participants familiar with the Gherkin language and the provided scenarios.

T3 corresponds to task *Run Scenarios* of the Testing Phase of the proposed process. Participants were expected to play the role of a Tester who is unfamiliar with the statechart design but is required to test the statechart behaviour on the basis of the provided scenarios.

T4 is a preparatory task to make the participants familiar with the textual and visual notation of the provided statechart.

Tasks T5 to T9 were designed to give the participants hands-on experience with the proposed techniques. This time, the participants play the role of a Software Engineer. The tasks were the following:

T5 aims at applying a full iteration over the statechart design and testing phase, focusing on the DbC technique. The participant finds herself at the position where new requirements need to be added to an existing (and previously tested) statechart. She starts by carrying out the task *Enrich Statechart with Contracts* to ensure that the timer value should never be negative and never exceed one hour. Next, the participant applies the *Manually test through UI* task to detect errors in the statechart design. Based on these errors she carries out the task *Define statechart* to make the statechart behaviour compatible with the added contract. The participant iterates until she is confident that the statechart is correctly defined.

T6 aims at applying another iteration over the statechart design and testing phase, focusing this time on the BDD technique. The participant is asked to add lamp functionality to the oven behaviour. To do so, she firsts carries out tasks *Define Scenarios* and *Run Scenarios* over the statechart. Based on the scenario test results, she carries out the task *Define statechart* to make the statechart behaviour compatible with the new scenarios. The participant iterates until she is confident that the statechart is correctly defined.

T7 provides more scenarios regarding lamp functionality to the participant. The participant needs to follow a similar process as in the previous task until she is confident that the statechart is correctly defined.

T8 aims at applying a full iteration over the statechart design and testing phase, focusing on the technique of property statecharts. First, participants are given the first property statechart depicted in Fig. 12 and iterate over the tasks *Define statechart*, *Manually test through UI* and

Monitor properties until they are confident that the statechart does not violate this property. Next, they are expected to carry out task *Define properties* to define the second property statechart of Fig. 12 and to iterate until they are confident that the statechart under test does not violate this property.

T9 aims at adding weight sensor functionality to the microwave oven, using all provided testing techniques. The participants have to define and run new scenarios for this functionality, and ensure that all existing contracts and properties remain satisfied while adding this new functionality. To do so, they use the *Manually test through UI* and *Run Scenarios* tasks, with both implicitly rely on *Monitor properties* and *Monitor contracts*.

At the end of each task, participants were asked to respond to a series of questions, some following a Likert rating scale, and others being open-ended in order to allow us to receive more detailed feedback. The questions aimed at assessing the usefulness and usability of the three proposed statechart testing and validation techniques. We present the analysis results of each technique separately below. We summarise the responses to each Likert-scale question in figures (e.g., Fig. 23) that combine a bar chart with a boxplot showing the minimum, first quartile, median, third quartile, and maximum values. In addition, a red vertical dashed line shows the mean value.

7.2 Evaluation of BDD

At the outset of the experiment, most participants indicated that they were either unaware of BDD, or that they were knowledgeable about the technique but never used it before (Fig. 23). During the experiment, the participants were asked to run existing BDD scenarios, as well as to provide and test more scenarios while extending the statechart with more functionality. At the end of the experiment, at least 9 out of 13 participants responded that they were either **very** or **extremely likely** to use the BDD technique for the purpose of supporting statechart *creation* (9 participants), *verification* (9 participants) and *modification* (11 participants), respectively.

Fig. 24 shows that, after having read BDD scenarios for the provided example, most participants found them to be **very** or **extremely easy** to understand, and **easy** or **very easy** to specify themselves. Fig. 25 (left) shows that participants were **very** or **extremely** confident that the oven controller satisfied the provided scenarios. After executing the scenarios with `sismic-bdd` their confidence increased even more, on average (Fig. 25 right).

Overall, participants were very positive about the BDD approach. For example, one participant found them "*useful for defining high-level tests, even non-programmers (eg. the client) can write them.*" Another participant answered that "*it is really user-friendly as there is no need to learn some specific language and we can just read the scenarios like user stories.*"

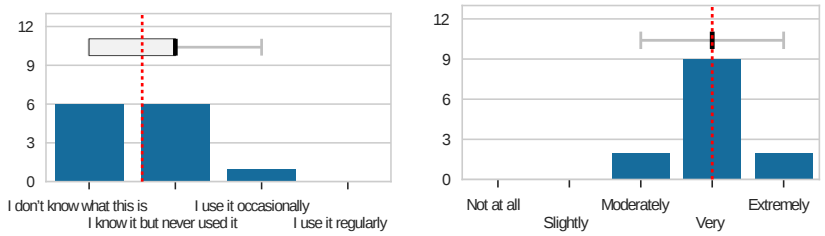


Fig. 24 Participant familiarity with BDD before the experiment (left), and likelihood of using BDD during modification of existing statecharts after the experiment (right).

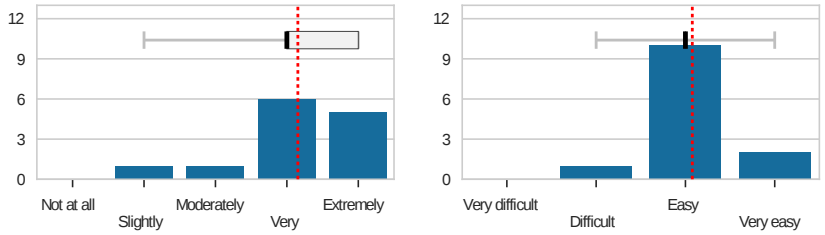


Fig. 25 Understandability (left) and ease of specification (right) of BDD scenarios.

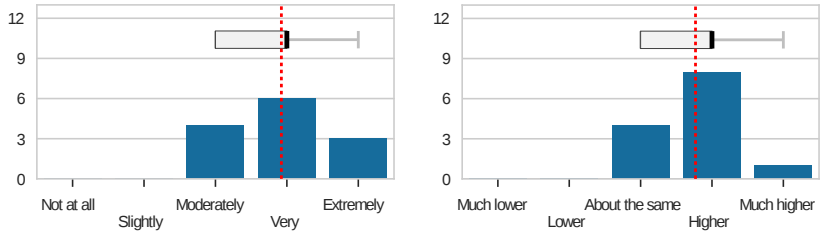


Fig. 26 Confidence in the specified BDD scenarios before their execution (left) and increase of confidence after their execution (right).

Related to the expressiveness, one participant believed the natural specification language to be "*suitable enough to describe most of the possible scenarios.*" Finally, one participant responded that "*even for 'easy' changes in the statecharts, related scenarios helped me to validate the correctness of my work.*"

From the negative side, some participants complained about the verbosity, lack of scalability and lack of exhaustiveness. For example, one participant found it "*hard to be exhaustive while specifying tests.*" Another one asked "*Does it scale up to complexity with more complex use cases?*" A third one answered that "*Using human-readable functional scenarios is too*

verbose. The real behaviour of the scenario is hidden behind the peculiarities of the 'human readable' language used."

7.3 Evaluation of DbC for statechart monitoring

At the outset of the experiment, most participants were knowledgeable about DbC, but only 3 participants had actually used it (Fig. 26 left). During the experiment, participants were asked to add new contracts to an existing statechart, to monitor the statechart behaviour with these contracts, and to modify the contracts while extending the statechart with new functionality. At the end of the experiment, at least 9 out of 13 participants responded that they were either **very** or **extremely likely** to use DbC for the purpose of *creating* (10 participants), *verifying* (10 participants) or *modifying* (9 participants) statecharts respectively.

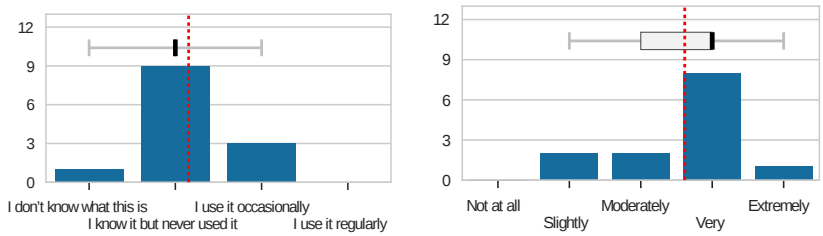


Fig. 27 Participant familiarity with DbC before the experiment (left), and likelihood of using DbC during modification of existing statecharts after the experiment (right).

Fig. 27 shows that most partipicants were **very** or **extremely** convinced about the usefulness of contract specification and monitoring during statechart design, and they found the implementation as provided by *Sismic* to be **convenient** or **very convenient**.

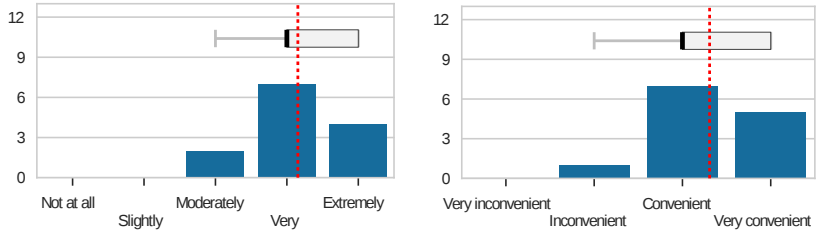


Fig. 28 Usefulness of statechart contracts (left), and convenience of their implementation in *Sismic* (right).

7.4 Evaluation of Property Statecharts

We did not ask participants about previous familiarity with the technique of property statecharts, as this technique was newly proposed for *Sismic*. During the experiment, the participants were asked to check the statechart for property violations on the basis of a property statechart that was provided to them. They were also asked to add and monitor another property statechart representing a different property.

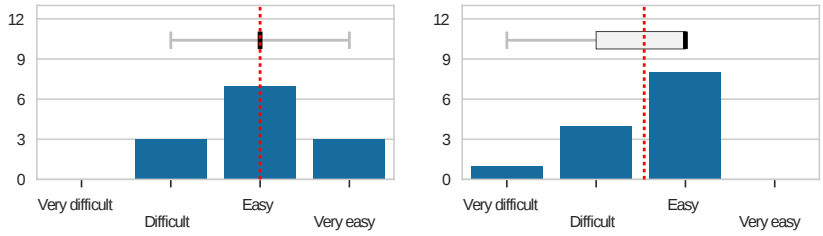


Fig. 29 Understandability (left) and ease of designing property statecharts (right).

Fig. 28 shows that most participants (10) found property statecharts to be **easy** or **very easy** to understand. On average, participants found it less easy to write property statecharts themselves. 5 out of 13 participants even found this **difficult** or **very difficult**. At the end of the experiment, most participants found property statecharts to be **very** useful or better (Fig. 29, left). Despite their perceived difficulty, 8 out of 13 participants responded that they were **very likely** to use it in the future for *modifying* existing statecharts.

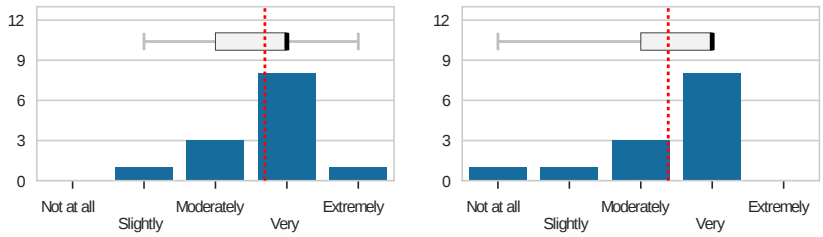


Fig. 30 Usefulness of property statecharts (left) and likelihood of using property statecharts during modification of existing statecharts (right).

7.5 Discussion

The controlled user study ended with an appreciation of the usefulness of each evaluated technique for testing and validating statecharts, and the ease of use of its implementation in *Sismic*. The received responses provide initial evidence that BDD scenarios and runtime monitoring of contracts and property statecharts are all beneficial during statechart design. Most participants indicated that they were (very) likely to use each of these techniques in the future for the purpose of creating new statecharts, or for verifying or modifying existing ones.

It is worthwhile to note that the large majority of the participants had never used BDD or DbC before participating in the study. Most of them also did not have a formal methods background. Despite this, they did not have a big problem in applying the proposed approaches for the purpose of statechart testing.

The *complementarity* of the statechart testing and validation techniques was highlighted by the participants in their responses. On the one hand, the ability of runtime monitoring of contracts and property statecharts were appreciated by one participant "*because it can capture several paths*": the properties and assertions that are expressed by property statecharts and contracts are verified at runtime, regardless of the followed execution path. Nevertheless, as indicated by another participant, it "*requires a large number of tests to be useful*". On the other hand, a participant found BDD scenarios to be "*suitable enough to describe most of the possible scenarios*", but agreed that it is "*hard to be exhaustive while specifying tests*". Indeed, BDD scenarios only test one execution path at a time, and being exhaustive would require a very large, even infinite number of scenarios. A similar remark could be made about the expressiveness of statechart contracts. Because they are defined on individual states, it is not possible to use them to express behavioural properties or invariants that involve visiting multiple states. Another limitation in expressiveness is caused by the absence of some kind of "memory", preventing contracts to express the temporality or causality required for some properties. Contracts should therefore be considered as a complementary technique to the mechanism of property statecharts. This discussion reveals the need of providing and combining a range of different and complementary techniques for testing and validating statecharts, as proposed by our method and associated tooling.

The implementation of *Sismic* that was used for the experiment did not yet provide any means for *visualising* statecharts. Being aware of this possible limitation, we assessed its effect on the ease of reading, designing and modifying statecharts. Fig. 30 and Fig. 31 reveal a slight tendency of participants preferring a visual over a textual representation. One of the participants commented that "*generally speaking the choice would depend on the person using it*" and we "*see benefit in providing both (or rather, not removing the textual option given the availability of a visual editor)*". We have taken into account this valuable remark, and provided support for

visualising statecharts using PlantUML, and editing statecharts graphically using the ASEME IDE.

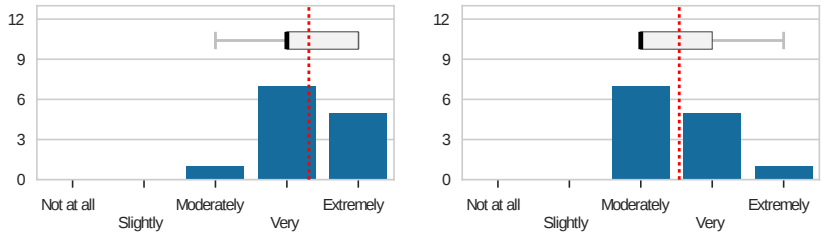


Fig. 31 Readability of a visual (left) or textual (right) statechart representation.

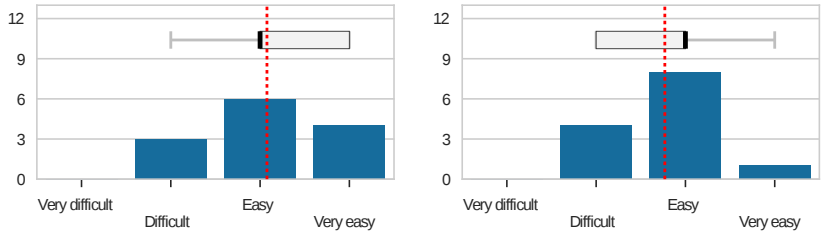


Fig. 32 Ease of designing a statechart using a visual (left) or textual (right) representation.

7.6 Threats

The controlled study suffers from several threats to validity. Therefore, the findings should merely be used as anecdotal evidence of the usefulness of the proposed process and techniques.

As a first threat to validity, the number of participants was fairly low, making it difficult to generalise the findings. Secondly, the selection of the participants based on convenience sampling introduced an inevitable bias: all participants were either students or researchers. The educational background of the participants may have played a role in the evaluation, and practitioners or company employees involved in statechart design might have another opinion.

In order to use the proposed method and *Sismic* tooling, a certain amount of programming experience is desirable, due to the fact that *Sismic* is a library as opposed to a full-fledged visual modeling environment. All selected participants had some amount of programming practice in Python

(ranging between less than 1 year and more than 5 years), and their self-assessed proficiency with Python ranged from poor to good.

Another threat relates to the complexity of the tasks that the participants were asked to carry out. Given the limited amount of available time, the examples and tasks provided to them were relatively simple. Hence, we cannot generalise the findings to more realistic designs that are likely to be more complex.

Because of the fairly low number of participants, our study did not include a control group. As a consequence, we cannot make any claims on how the proposed techniques compare to other techniques (or using none at al) for statechart testing and validation. In follow-up work, we plan to evaluate the actual impact of using the proposed techniques and tools on the quality of statechart designs, as well as on the productivity of the designers and testers.

8 Related Work

Related tools. Many tools exist for specifying executable statecharts and generating code from them (e.g., *StateMate*, *Rhapsody*, *Stateflow*, *Yakindu*, *visualSTATE*, and many more). Most of these tools are commercial, and offer a complete modelling environment. *Sismic* differs from this in two ways. First of all, it is provided as a *library* to facilitate its use as part of other applications. Secondly, it is provided as a fully documented and modular open source research prototype, so that it can be used freely as a platform for carrying out research experiments. While many tools (or plugins for them) provide support for test-driven design, we are not aware of any tool providing the combination of testing techniques provided by *Sismic*. Within the open source realm, two tools are nevertheless worthwhile mentioning. *Papyrus* (`www.eclipse.org/papyrus/`), combined with *Moka*, is an integrated modelling environment in Eclipse supporting simulation and execution of UML models. It does not provide built-in support for statechart testing as in *Sismic*, but plug-ins could be developed to achieve this. *AutoFOCUS 3* (`af3.fortiss.org`) is an integrated open source modelling tool that supports requirements analysis, simulation and testing of models, formal model checking and verification, architectural design and optimisation of software and hardware components, and design space exploration [3]. While being very complete, it is different from *Sismic* in that the behavioural models are expressed as finite state automata rather than full-fledged hierarchical statecharts.

Unit testing of statechart models. Dietrich et al. [14] applied unit testing to UML statecharts, and implemented it in a model simulation tool called *Syntony*. Test cases were defined using UML sequence diagrams, providing scenarios that are verified over the statechart under test. The Yakindu statechart tool for Java (`www.statechart.org`) also provides support for

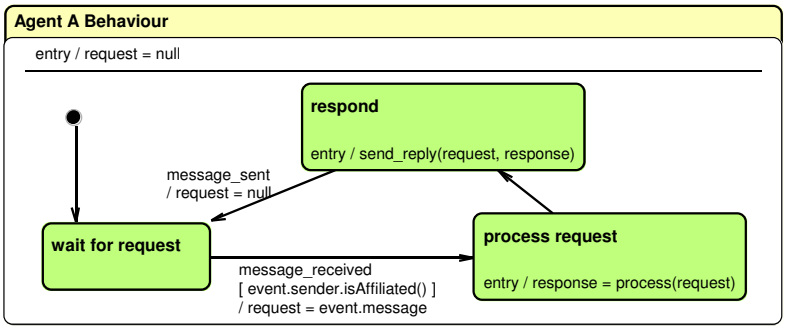
unit testing using SCUnit, a framework that supports writing tests over statecharts, and running these tests with the JUnit testing framework.

BDD for statechart modelling. We found very little related work on the use of BDD at a modelling level. Lazar et al. [35] proposed bUML, a tool to combine BDD with model-driven development based on a specific UML profile, and compliant with the fUML action language. While the authors propose a visual UML syntax for expressing BDD scenarios, they only apply it to planning and monitoring of project progress, rather than using it for testing behavioural models like UML statecharts, as is the case in *Sismic*.

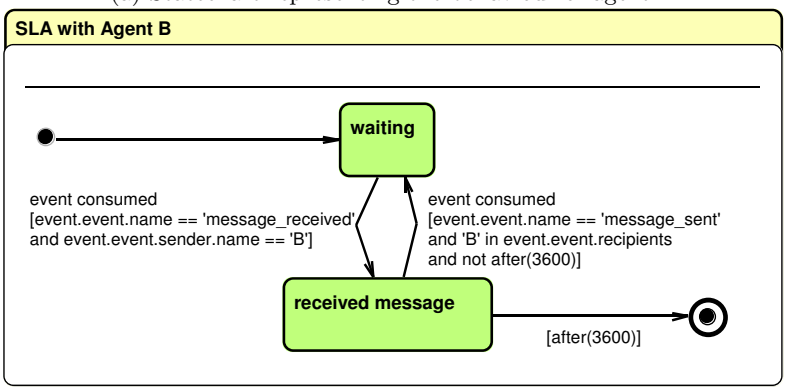
DbC for statechart modelling. At the level of UML models, DbC has mainly be used for class diagrams. Cabot [8] used the UMLtoCSP tool to transform such contracts into a constraint satisfaction problem, that is fed to a constraint solver. Gogolla implemented support for validating such contracts through the USE tool [25]. This tool has been extended with contract support for protocol state machines [28] and sequence diagrams [26]. Cariou et al. [9] applied contracts at the metamodel level to verify at runtime whether the statechart execution semantics is respected. Cimatti et al. [10] used component contracts to specify the behaviour of interacting components, and combined this with a temporal logic framework to formally verify the contracts using the OCRA tool and the NuSMV symbolic model checker. It has a plugin for the AutoFOCUS3 and the CHESS modelling tool.

Recently, the technique of *smart contracts* has emerged as a way to specify programs that enforce the application of rules to govern transactions [13], to safeguard contractual clauses [34], and to define quality of service (QoS) characteristics (e.g., performance, availability, security) [7]. For example, smart contracts have been proposed for supporting cryptocurrency protocols [13,34], as well as executable Service Level Agreements (SLAs) for the smart grid [7]. The proposed mechanism of property statecharts can be used as a way to realise smart contracts. As an example, assume that there is an agent A that signs a Service Level Agreement (SLA) with agent B. The SLA dictates that whenever A receives a request from B, then A must reply within 1 hour. The statecharts in Fig. 32 depict the behaviour of agent A and the property statechart monitoring the agent's SLA. The agent receives requests and if they are made by affiliates it processes them and subsequently responds. The property statechart monitoring the agent's SLA with agent B starts in the `waiting` state. If a request from agent B is received by agent A and more than one hour passes before a response has been issued to B then the contract is terminated (fails).

Automatic generation of tests and contracts. Ernst et al. [19] proposed to automate the generation of contracts over programs, and to detect possible inconsistencies or incompleteness in the specified contracts. To this extent,



(a) Statechart representing the behaviour of agent A



(b) Property statechart monitoring the SLA with another agent B.

Fig. 33 Example of statecharts for smart contracts between two communicating agents A and B. The models have been edited using the ASEME IDE statechart editor.

he developed the Daikon tool that automatically detects likely invariants from dynamic program executions. In a similar vein, Meyer [40] proposed to deduce automated tests from contract specifications. This reduces the burden of needing to write many tests manually, and facilitates checking correctness of a system. It would be beneficial to apply the same idea to generate statechart tests both from the contracts and the behavioural properties defined over the statechart and its components, especially in combination with the use of *mutation testing* to increase the quality of an existing test suite [22,49], and *concolic testing* (a combination of concrete and symbolic execution) to generate new test cases to achieve higher coverage [45].

Formal verification and runtime verification. Approaches based on formal verification and model checking allow to verify properties (e.g., related to reachability, liveness, fairness and safety) over the system under study (in our case, a statechart model) [4]. Such techniques often rely on a model checker tool, and need to cope with problems related to space explosion. Formalisms based on *temporal logics* allow to specify properties qualified in

terms of time, and rely on a model checker to verify such properties. For example, Gnesi et al. [24] presented JACK, a model checker that verifies correctness properties over UML statecharts. These properties are expressed in a variant of branching time temporal logic, and a labeled transition system (derived from the statechart) is checked against these requirements. Another family of verification approaches is the one based on some variant of finite automata (a.k.a. finite state machines or FSMs). This includes labeled transition systems, timed automata, hybrid automata, and counter automata. They have the advantage of being more similar to the statechart language (since they are also based on states and transitions), hence reducing the mental gap to express properties in these formalisms. Among many others, the LTSA tool, based on labeled transition systems, has been proposed to analyse the behaviour of concurrent systems [38]. While their usefulness seems without doubt, their usability has been criticised, and different attempts have been made to increase the usability by non-logicians [18,11,6].

Runtime verification is based on similar techniques (including regular expressions, temporal logics, state machines and rule-based programming [23]), but aims at monitoring observable behaviours or properties over an executing system. Runtime verification checks for violation of properties at runtime, at the expense of providing less coverage than formal verification approaches [37]. While numerous approaches have used state machines or related mechanisms to monitor the execution of programs, much less research has focused on monitoring of executable statechart models. Drusinsky [16, 17] proposed to monitor violations of temporal logic assertions aver statecharts at runtime. The notion of property statecharts explored in this article is different, in that it offers the full expressive power of statecharts to monitor properties over statecharts. Because arbitrary Python code can be used (e.g., in transition actions), property statecharts are Turing complete. Nevertheless, due to the "fail-fast" approach adopted by *Sismic*'s interpreter for their verification, property statecharts are mainly intended to check for the presence of undesirable behaviour (safety properties), i.e., properties that can be checked on a (finite) prefix of a (possibly infinite) execution trace. While it is technically possible to use property statecharts to express liveliness properties (something desirable *eventually* happens), this would require additional support for their verification.

9 Conclusion

This article presented a new method for testing and validating executable statecharts. The method is supported by *Sismic*, an open source research prototype tool that we developed specifically for this purpose. The Sismic method enhances traditional statechart design with a range of techniques that have already proven their usefulness for source code development. All of these techniques can be combined easily to test and validate statechart designs.

A straightforward way of testing statecharts relies on writing unit tests, but this introduces an unnecessary technical gap of needing to write these tests in an underlying programming language. The technique of Behaviour-Driven Development (BDD) overcomes this limitation by allowing to specify scenarios of desired functional behaviour in a semi-formal natural language, and to use these scenarios as executable functional tests over the statechart. This approach has the advantage of only needing a minimum amount of mapping code.

We also provided two techniques for runtime verification of statecharts. The first one consists of applying the principle of Design by Contract (DbC) at the level of statecharts, allowing to specify preconditions, postconditions and invariants on states and transitions, and monitoring violations these contracts during statechart execution. The second technique consists of specifying behavioural properties as statecharts themselves, allowing the statechart under test to be monitored at runtime for violations of these properties.

We evaluated the proposed method, and more in particular the usefulness and usability of the proposed techniques through a controlled user study conducted with thirteen participants. All techniques were considered to be useful and complementary. Participants reported that they were likely to use each of the proposed techniques for creating new statecharts, or for verifying or modifying existing ones. Participants also indicated that the implementation of the techniques in *Sismic* was easy to use.

Acknowledgments

We express our gratitude to Jordi Cabot, Simon Van Mierlo, Gauvain Devillez and Mathieu Goeminne, and several anonymous reviewers for providing comments on earlier versions of this article.

References

1. Object Management Group. *Object Constraint Language (OCL). Version 2.4*, Feb. 2014.
2. Object Management Group. *Unified Modeling Language (UML), Superstructure. Version 2.5*, Mar. 2015.
3. V. Aravantinos, S. Voss, S. Teufl, F. Hölzl, and B. Schätz. AutoFOCUS 3: Tooling concepts for seamless, model-based development of embedded systems. In *Int'l Workshop on Model-based Architecting of Cyber-physical and Embedded Systems and Int'l Workshop on UML Consistency Rules*, volume 1508 of *CEUR Workshop Proceedings*, pages 19–26. CEUR-WS.org, 2015.
4. C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT Press, 2008.
5. K. Beck. *Test-Driven Development by Example*. Addison-Wesley, 2002.
6. I. Beer, S. Ben-David, C. Eisner, D. Fisman, A. Gringauze, and Y. Rodeh. The temporal logic sugar. In *Int'l Conf. Computer Aided Verification (CAV)*, pages 363–367. Springer, 2001.

7. C. Bunse, S. Klingert, and T. Schulze. Greenslas: Supporting energy-efficiency through contracts. In *International Workshop on Energy Efficient Data Centers*, pages 54–68. Springer, 2012.
8. J. Cabot, R. Clarisó, and D. Riera. On the verification of UML/OCL class diagrams using constraint programming. *J. Systems and Software*, 93:1 – 23, 2014.
9. E. Cariou, C. Ballagny, A. Feugas, and F. Barbier. Contracts for model execution verification. In *European Conf. Modelling Foundations and Applications (ECMFA)*, volume 6698 of *Lect. Notes in Computer Science*, pages 3–18. Springer, 2011.
10. A. Cimatti and S. Tonetta. Contracts-refinement proof system for component-based embedded systems. *Science of Computer Programming*, 97:333–348, 2015.
11. J. C. Corbett, M. B. Dwyer, J. Hatcliff, and Robby. A language framework for expressing checkable properties of dynamic software. In *Int'l SPIN Model Checking and Software Verification Workshop*, volume 1885 of *Lect. Notes in Computer Science*, pages 205–223. Springer, 2000.
12. M. Cossentino, S. Gaglio, A. Garro, and V. Seidita. Method fragments for agent design methodologies: from standardisation to research. *International Journal of Agent-Oriented Software Engineering*, 1(1):91–121, 2007.
13. K. Delmolino, M. Arnett, A. Kosba, A. Miller, and E. Shi. Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab. In *International Conference on Financial Cryptography and Data Security*, pages 79–94. Springer, 2016.
14. I. Dietrich, F. Dressler, W. Dulz, and R. German. Validating UML simulation models with model-level unit tests. In *Int'l Conf. Simulation Tools and Techniques (SIMUTools)*, 2010.
15. B. P. Douglas. *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns*. Addison-Wesley, 1999.
16. D. Drusinsky. Semantics and runtime monitoring of TLCharts: Statechart automata with temporal logic conditioned transitions. *Electronic Notes in Theoretical Computer Science*, 113:3 – 21, 2005. Proceedings of the Fourth Workshop on Runtime Verification (RV 2004).
17. D. Drusinsky. *Modeling and Verification Using UML Statecharts*. Elsevier Science, 2006.
18. M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *Int'l Conf. Software Engineering*, pages 411–420. ACM , 1999.
19. M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1-3):35–45, 2007.
20. S. Esmaeilsabzali, N. A. Day, J. M. Atlee, and J. Niu. Deconstructing the semantics of big-step modelling languages. *Requirements Engineering*, 15(2):235–265, 2010.
21. H. Estler, C. A. Furia, M. Nordio, M. Piccioni, and B. Meyer. Contracts in practice. In *Int'l Symp. Formal Methods (FM)*, volume 8442 of *Lect. Notes in Computer Science*, pages 230–246. Springer, 2014.
22. S. C. P. F. Fabbri, J. C. Maldonado, T. Sugeta, and P. C. Masiero. Mutation testing applied to validate specifications based on statecharts. In *Int'l Symp. Software Reliability Engineering (ISSRE)*, pages 210–219. IEEE Computer Society, 1999.

23. Y. Falcone, K. Havelund, and G. Reger. A tutorial on runtime verification. *Engineering Dependable Software Systems*, 34:141–175, 2013.
24. S. Gnesi, D. Latella, and M. Massink. Model checking UML statechart diagrams using JACK. In *Int'l Symp. High-Assurance Systems Engineering (HASE)*, pages 46–55. IEEE Computer Society, 1999.
25. M. Gogolla, F. Büttner, and M. Richters. USE: A UML-based specification environment for validating UML and OCL. *Science of Computer Programming*, 69(1-3):27–34, 2007.
26. M. Gogolla, L. Hamann, F. Hilken, and M. Sedlmeier. Modeling behavior with interaction diagrams in a UML and OCL tool. In *Behavior Modeling - Foundations and Applications, BM-FA 2009-2014, Revised Selected Papers*, volume 6368 of *Lect. Notes in Computer Science*, pages 31–58. Springer, 2015.
27. H. Gomaa. *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*. Addison Wesley, 2004.
28. L. Hamann, O. Hofrichter, and M. Gogolla. On integrating structure and behavior modeling with OCL. In *Int'l Conf. Model Driven Engineering Languages and Systems*, volume 7590 of *Lect. Notes in Computer Science*, pages 235–251. Springer, 2012.
29. D. Harel. On visual formalisms. *Comm. ACM*, 31(5):514–530, 1988.
30. D. Harel and E. Gery. Executable object modeling with statecharts. *IEEE Computer*, 30(7):31–42, July 1997.
31. D. Harel and H. Kugler. *The Rhapsody Semantics of Statecharts (or, On the Executable Core of the UML)*, volume LNCS 3147. Springer, 2004.
32. D. Harel and A. Naamad. The STATEMATE semantics of statecharts. *ACM Trans. Softw. Eng. Methodol.*, 5(4):293–333, Oct. 1996.
33. B. Henderson-Sellers and J. Ralyté. Situational method engineering: state-of-the-art review. *Journal of Universal Computer Science*, 16(3):424–478, 2010.
34. F. Idelberger, G. Governatori, R. Riveret, and G. Sartor. Evaluation of logic-based smart contracts for blockchain systems. In *International Symposium on Rules and Rule Markup Languages for the Semantic Web*, pages 167–183. Springer, 2016.
35. I. Lazar, S. Motogna, and B. Parv. Behaviour-driven development of foundational UML components. *Electronic Notes in Theoretical Computer Science*, 264(1):91 – 105, 2010. Int'l Workshop on Formal Engineering approaches to Software Components and Architectures (FESCA).
36. G. T. Leavens and Y. Cheon. Design by Contract with JML. Technical report, Iowa State University, 2006.
37. M. Leucker and C. Schallhart. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 78(5):293 – 303, 2009. The 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS'07).
38. J. Magee. Behavioral analysis of software architectures using LTSA. In *Int'l Conf. Software Engineering*, pages 634–637. ACM, 1999.
39. B. Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, 1992.
40. B. Meyer. Contract-driven development. In *Int'l Conf. Fundamental Approaches to Software Engineering (FASE)*, volume 4422 of *Lect. Notes in Computer Science*, page 11. Springer, 2007.
41. D. North. Behavior modification: The evolution of behavior-driven development. *Better Software*, 2006.

42. OMG. Software and Systems Process Engineering Meta-Model Specification. Version 2.0. Technical Report OMG Document Number: formal/2008-04-01, Object Management Group, 2008.
43. Y. Pei, C. A. Furia, M. Nordio, Y. Wei, B. Meyer, and A. Zeller. Automated fixing of programs with contracts. *IEEE Trans. Soft. Eng.*, 40(5):427–449, 2014.
44. M. Samek. *Practical UML Statecharts in C/C++: Event-Driven Programming for Embedded Systems*. CRC Press, second edition, 2008.
45. K. Sen. Concolic testing. In *Int'l Conf. Automated Software Engineering*, pages 571–572. ACM, 2007.
46. N. Spanoudakis and P. Moraitis. The Agent Modeling Language (AMOLA). In D. Dochev, M. Pistore, and P. Traverso, editors, *Artificial Intelligence: Methodology, Systems, and Applications*, volume 5253 of *Lecture Notes in Computer Science*, pages 32–44. Springer Berlin Heidelberg, 2008.
47. E. Syriani, H. Vangheluwe, R. Mannadiar, C. Hansen, S. Van Mierlo, and H. Ergin. AToMPM: A web-based modeling environment. In *Joint Proceedings of MODELS'13 Invited Talks, Demonstration Session, Poster Session, and ACM Student Research Competition*, volume 1115. CEUR Workshop Proccedings, 2013.
48. A. Topalidou-Kyniazopoulou, N. I. Spanoudakis, and M. G. Lagoudakis. A CASE Tool for Robot Behavior Development. In X. Chen, P. Stone, L. E. Sucar, and T. Zant, editors, *RoboCup 2012: Robot Soccer World Cup XVI*, volume 7500 of *Lecture Notes in Computer Science*, pages 225–236. Springer Berlin Heidelberg, 2013.
49. M. Trakhtenbrot. New mutations for evaluation of specification and implementation levels of adequacy in testing of statecharts models. In *Testing: Academic and Industrial Conference Practice and Research Techniques (MUTATION)*, pages 151–160, Sept. 2007.
50. M. Wynne and A. Hellesoy. *The Cucumber Book: Behaviour-Driven Development for Testers and Developers*. Pragmatic Bookshelf, 2012.