Sismic Documentation

Release 0.22.11

Alexandre Decan

Jun 08, 2017

Overview

CHAPTER 1

About

Sismic is a recursive acronym that stands for *Sismic Interactive Statechart Model Interpreter and Checker*.

The Sismic library for Python (version 3.4 or higher) is mainly developed by Alexandre Decan at the University of Mons.

Sismic is released publicly under the GNU Lesser General Public Licence version 3.0 (LGPLv3).

Sismic provides a set of tools to define, validate, simulate, execute and test statecharts. Statecharts are a well-known visual modeling language for representing the executable behavior of complex reactive event-based systems.

CHAPTER 2

Features

Sismic provides the following features:

- An easy way to define and to import statecharts, based on the human-friendly YAML markup language
- A statechart interpreter offering a discrete, step-by-step, and fully observable simulation engine
- Synchronous and asynchronous simulation, in real time or simulated time
- Support for communication between statecharts and co-simulation
- Built-in support for expressing actions and guards using regular Python code, can be easily extended to other programming languages
- A design-by-contract approach for statecharts: contracts can be specified to express invariants, sequential conditions, pre- and postconditions on states and transitions
- Predefined step definitions and utilities (including test coverage) to support behavior-driven development
- A unit testing framework for statecharts, including generation of test scenarios

The semantics of the statechart interpreter is based on the specification of the SCXML semantics (with a few exceptions), but can be easily tuned to other semantics. Sismic statecharts provides full support for the majority of the UML 2 statechart concepts:

- simple states, composite states, orthogonal (parallel) states, initial and final states, shallow and deep history states
- state transitions, guarded transitions, automatic (eventless) transitions, internal transitions
- statechart (scoped) variables and their initialisation
- state entry and exit actions, transition actions
- internal and external parametrized events

Installation

Using pip

Sismic can be installed using `pip` as usual: `pip install sismic`. This will install the latest stable version. Sismic requires Python >=3.4. You can isolate Sismic installation by using virtual environments:

1. Get the tool to create virtual environments: `pip install virtualenv`
2. Create the environment: `virtualenv -p python3.4 env`
3. Jump into: `source env/bin/activate`
4. Install Sismic: `pip install sismic`

The development version can also be installed directly from its git repository: `pip install git+git://github.com/AlexandreDecan/sismic.git@devel`

From GitHub

You can also install Sismic from its repository by cloning it. The development occurs in the *devel* branch, the latest stable distributed version is in the *master* branch.

1. Get the tool to create virtual environments: `pip install virtualenv`
2. Create the environment: `virtualenv -p python3.4 env`
3. Jump into: `source env/bin/activate`
4. Clone the repository: `git clone https://github.com/AlexandreDecan/sismic`
5. Install dependencies: `pip install -r requirements.txt`

Sismic is now available from the root directory. Its code is in the *sismic* repository. The documentation can be built from the *docs* directory using `make html`.

Tests are available both for the code and the documentation:

- `make doctest` in the *docs* directory (documentation tests)
- `python -m unittest discover` from the root directory (code tests)

Statecharts definition

About statecharts

Statecharts are a well-known visual language for modeling the executable behavior of complex reactive event-based systems. They were invented in the 1980s by David Harel, and have gained a more widespread adoption since they became part of the UML modeling standard.

Statecharts offer more sophisticated modeling concepts than the more classical state diagrams of finite state machines. For example, they support hierarchical composition of states, orthogonal regions to express parallel execution, guarded transitions, and actions on transitions or states. Different flavours of executable semantics for statecharts have been proposed in the literature and in existing tools.

Defining statecharts in YAML

Because Sismic is supposed to be independent of a particular visual modeling tool, and easy to integrate in other programs without requiring the implementation of a visual notation, statecharts are expressed using YAML, a human-friendly textual notation (the alternative of using something like SCXML was discarded because its notation is too verbose and not really "human-readable".)

See also:

This section explains how the elements that compose a valid statechart in Sismic can be defined using YAML. If you are not familiar with YAML, have a look at YAML official documentation.

Statechart

The root of the YAML file **must** declare a statechart:

```
statechart:
  name: Name of the statechart
  description: Description of the statechart
  root state:
    [...]
```

The *name* and the *root state* keys are mandatory, the *description* is optional. The *root state* key contains a state definition (see below). If specific code needs to be executed during initialization of the statechart, this can be specified using *preamble*. In this example, the code is written in Python.

```
statechart:
  name: statechart containing initialization code
  preamble: x = 1
```

Code can be written on multiple lines as follows:

```
preamble: |
  x = 1
  y = 2
```

States

A statechart must declare a root state. Each state consist of at least a mandatory *name*. Depending on the state type, other optional fields can be declared.

```
statechart:
  name: with state
  root state:
    name: root
```

Entry and exit actions

For each declared state, the optional *on entry* and *on exit* fields can be used to specify the code that has to be executed when entering and leaving the state:

```
- name: s1
  on entry: x += 1
  on exit: |
```

```
    x -= 1
    y = 2
```

Final states

A *final state* can be declared by specifying *type: final*:

```
- name: s1
  type: final
```

Shallow and deep history states

History states can be declared as follows:

- *type: shallow history* to declare a *shallow history* state;
- *type: deep history* to declare a *deep history* state.

```
- name: history state
  type: shallow history
```

A history state can optionally declare a default initial memory using *memory*. Importantly, the *memory* value **must** refer to a parent's substate.

```
- name: history state
  type: deep history
  memory: s1
```

See also:

We refer to the semantics of UML for the difference between both types of histories.

Composite states

A state that is neither a final state nor a history state can contain nested states. Such a state is commonly called a *composite state*.

```
- name: composite state
  states:
    - name: nested state 1
    - name: nested state 2
      states:
        - name: nested state 2a
```

A composite state can define its initial state using *initial*.

```
- name: composite state
  initial: nested state 1
  states:
    - name: nested state 1
    - name: nested state 2
      initial: nested state a2
      states:
        - name: nested state 2a
```

Note: Unlike UML, but similarly to SCXML, Sismic does not explicitly represent the concept of *region*. A region is essentially a logical set of nested states, and thus can be viewed as a specialization of a composite state.

Orthogonal states

Orthogonal states (sometimes referred as *parallel states*) allow to specify multiple nested statecharts running in parallel. They must declare their nested states using *parallel states* instead of *states*.

```
statechart:
  name: statechart containing multiple orthogonal states
  initial state:
    name: processes
    parallel states:
      - name: process 1
      - name: process 2
```

Transitions

Transitions between states, compound states and parallel states can be declared with the *transitions* field. Transitions typically specify a target state using the *target* field:

```
- name: state with transitions
  transitions:
    - target: other state
```

Other optional fields can be specified for a transition: a *guard* (a Boolean expression that will be evaluated to determine if the transition can be followed), an *event* (name of the event that will trigger the transition), an *action* (code that will be executed if the transition is processed). Here is a full example of a transition specification:

```
- name: state with an outgoing transition
  transitions:
    - target: some other state
      event: click
      guard: x > 1
      action: print('Hello World!')
```

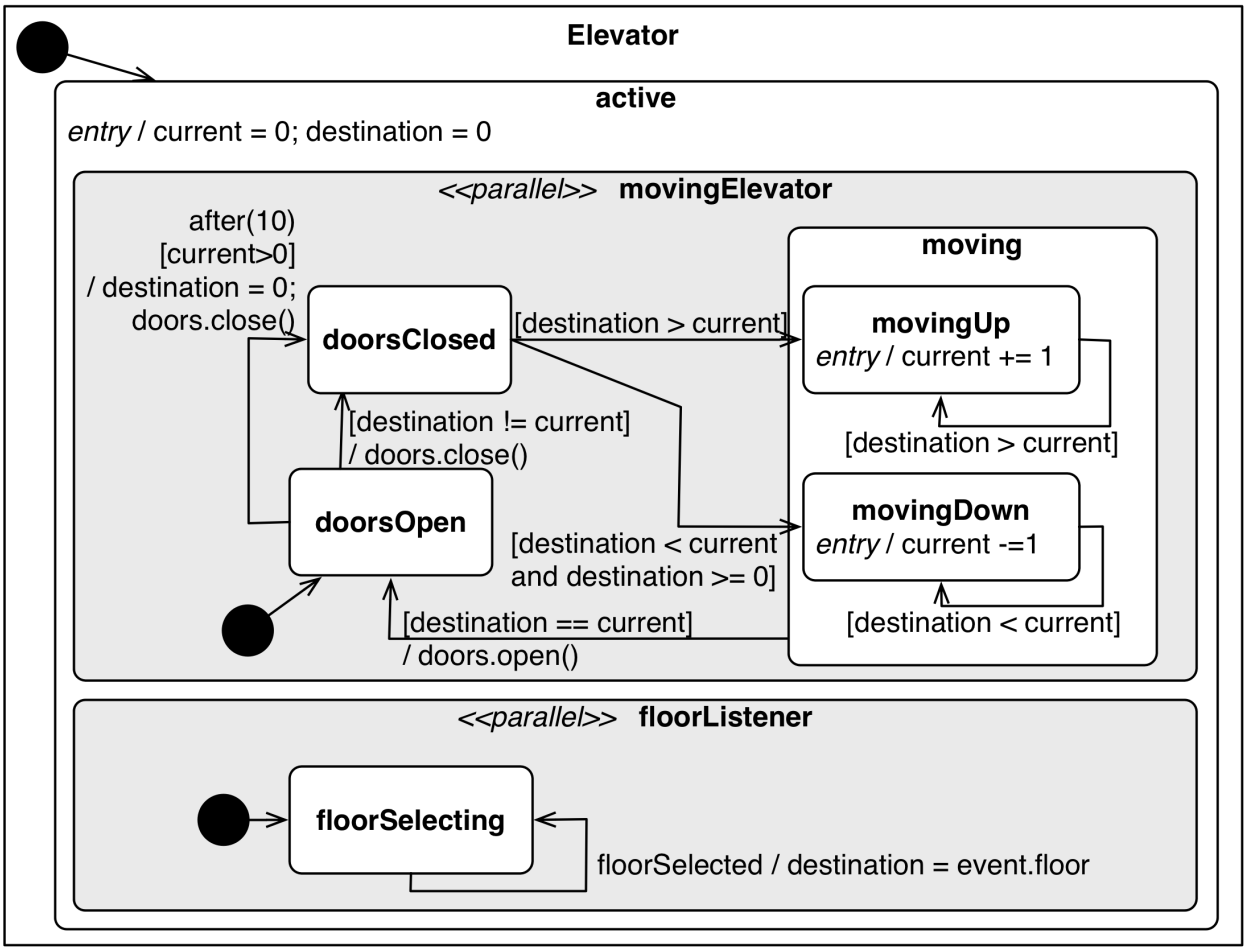
One type of transition, called an *internal transition*, does not require to declare a *target*. Instead, it **must** either define an event or define a guard to determine when it should become active (otherwise, infinite loops would occur during simulation or execution).

Notice that such a transition does not trigger the *on entry* and *on exit* of its state, and can thus be used to model an *internal action*.

Statechart examples

Elevator

The Elevator statechart is one of the running examples in this documentation. Its visual description (currently not supported by Sismic) could look as follows:



The corresponding YAML description is given below.

```
statechart:
  name: Elevator
  preamble: |
    current = 0
    destination = 0
    doors_open = True
  root state:
    name: active
    parallel states:
      - name: movingElevator
        initial: doorsOpen
        states:
          - name: doorsOpen
            transitions:
              - target: doorsClosed
                guard: destination != current
                action: doors_open = False
              - target: doorsClosed
                guard: after(10) and current > 0
                action: |
                  destination = 0
                  doors_open = False
          - name: doorsClosed
            transitions:
```

```
            - target: movingUp
              guard: destination > current
            - target: movingDown
              guard: destination < current and destination >= 0
        - name: moving
          transitions:
            - target: doorsOpen
              guard: destination == current
              action: doors_open = True
          states:
            - name: movingUp
              on entry: current = current + 1
              transitions:
                - target: movingUp
                  guard: destination > current
            - name: movingDown
              on entry: current = current - 1
              transitions:
                - target: movingDown
                  guard: destination < current
    - name: floorListener
      initial: floorSelecting
      states:
        - name: floorSelecting
          transitions:
            - target: floorSelecting
              event: floorSelected
              action: destination = event.floor
```

Other examples

Some other examples can be found in the Git repository of the project, in docs/examples.

Importing and validating statecharts

The *Statechart* class provides several methods to construct, to query and to manipulate a statechart. A YAML definition of a statechart can be easily imported to a *Statechart* instance. The module *sismic.io* provides a convenient loader *import_from_yaml()* which takes a textual YAML definition of a statechart and returns a *Statechart* instance.

sismic.io.**import_from_yaml**(*statechart: typing.Iterable[str], ignore_schema: bool = False, ignore_validation: bool = False*) → sismic.model.statechart.Statechart

Import a statechart from a YAML representation.

Unless specified, the structure contained in the YAML is validated against a predefined schema (see *sismic.io.SCHEMA*), and the resulting statechart is validated using its *validate()* method.

Parameters

- **statechart** – string or any equivalent object
- **ignore_schema** – set to *True* to disable yaml validation.
- **ignore_validation** – set to *True* to disable statechart validation.

Returns a *Statechart* instance

For example:

```
from sismic import io, model

with open('examples/elevator/elevator.yaml') as f:
    statechart = io.import_from_yaml(f)
    assert isinstance(statechart, model.Statechart)
```

The parser performs an automatic validation against some kind of YAML schema to prevent erroneous keys (see below). It also does several other checks using statechart's *validate* method.

See also:

While statecharts can be defined in YAML, they can be defined in pure Python too. Moreover, *Statechart* instances exhibit several methods to query and manipulate statecharts (e.g.: *rename_state()*, *rotate_transition()*, *copy_from_statechart()*, etc.). Consider looking at *Statechart* API for more information.

YAML validation schema

See schema library for more information about the semantic.

```
class SCHEMA:
    contract = {schema.Or('before', 'after', 'always', 'sequentially'): schema.
↪Use(str)}

    transition = {
        schema.Optional('target'): schema.Use(str),
        schema.Optional('event'): schema.Use(str),
        schema.Optional('guard'): schema.Use(str),
        schema.Optional('action'): schema.Use(str),
        schema.Optional('contract'): [contract],
    }

    state = dict()  # type: Dict
    state.update({
        'name': schema.Use(str),
        schema.Optional('type'): schema.Or('final', 'shallow history', 'deep history
↪'),
        schema.Optional('on entry'): schema.Use(str),
        schema.Optional('on exit'): schema.Use(str),
        schema.Optional('transitions'): [transition],
        schema.Optional('contract'): [contract],
        schema.Optional('initial'): schema.Use(str),
        schema.Optional('parallel states'): [state],
        schema.Optional('states'): [state],
    })

    statechart = {
        'statechart': {
            'name': schema.Use(str),
            schema.Optional('description'): schema.Use(str),
            schema.Optional('preamble'): schema.Use(str),
            'root state': state,
        }
    }
```

Statecharts execution

Statechart semantics

The module `interpreter` contains an `Interpreter` class that interprets a statechart mainly following the SCXML 1.0 semantics. In particular, eventless transitions are processed *before* transitions containing events, internal events are consumed *before* external events, and the simulation follows a inner-first/source-state and run-to-completion semantics.

The main difference between SCXML and Sismic's default interpreter resides in how multiple transitions can be triggered simultaneously. This may occur for transitions in orthogonal/parallel states, or when transitions declaring the same event have guards that are not mutually exclusive.

Simulating the simultaneous triggering of multiple transitions is problematic, since it implies to make a non-deterministic choice on the order in which the transitions must be processed, and on the order in which the source states must the exited and the target states must be entered. The UML 2.5 specification explicitly leaves this issue unresolved, thereby delegating the decision to tool developers:

> "Due to the presence of orthogonal Regions, it is possible that multiple Transitions (in different Regions) can be triggered by the same Event occurrence. The **order in which these Transitions are executed is left undefined**." — UML 2.5 Specification

The SCXML specification addresses the issue by using the *document order* (i.e., the order in which the transitions appear in the SCXML file) as the order in which (non-parallel) transitions should be processed.

> "If multiple matching transitions are present, take the **first in document order**." — SCXML Specification

From our point of view, this solution is not satisfactory. The execution should not depend on the (often arbitrary) order in which items happen to be declared in some document, in particular when there many be many different ways to construct or import a statechart.

Other statechart tools do not even define any order on the transitions in such situations:

> "Rhapsody detects such cases of nondeterminism during code generation and **does not allow them**. The motivation for this is that the generated code is intended to serve as a final implementation and for most embedded software systems such nondeterminism is not acceptable." — The Rhapsody Semantics of Statecharts

We decide to follow Rhapsody and to raise an error (in fact, a `NonDeterminismError`) if such cases of nondeterminism occur during the execution. Notice that this only concerns multiple transitions in the same composite state, not in parallel states.

When multiple transitions are triggered from within distinct parallel states, the situation is even more intricate. According to the Rhapsody implementation:

> "The order of firing transitions of orthogonal components is not defined, and depends on an arbitrary traversal in the implementation. Also, the actions on the transitions of the orthogonal components are **interleaved in an arbitrary way**." — The Rhapsody Semantics of Statecharts

SCXML again circumvents this problem by using the *document order*.

> "enabledTransitions will contain multiple transitions only if a parallel state is active. In that case, we may have one transition selected for each of its children. [...] If multiple states are active (i.e., we are in a parallel region), then there may be multiple transitions, one per active atomic state (though some states may not select a transition.) In this case, the transitions are taken **in the document order of the atomic states** that selected them." — SCXML Specification

Again, Sismic does not agree with SCXML on this, and instead defines that multiple orthogonal/parallel transitions should be processed in a decreasing source state depth order. This is perfectly coherent with our aforementioned

inner-first/source-state semantics, as "deeper" transitions are processed before "less nested" ones. In case of ties, the lexicographic order of the source state names will prevail.

Note that in an ideal world, orthogonal/parallel regions should be independent, implying that *in principle* such situations should not arise ("*the designer does not rely on any particular order for event instances to be dispatched to the relevant orthogonal regions*", UML specification). In practice, however, it is often desirable to allow such situations.

Using *Interpreter*

An `Interpreter` instance is constructed upon a `Statechart` instance and an optional callable that returns an `Evaluator`. This callable must accept an interpreter and an initial execution context as input (see *Include code in statecharts*). If not specified, a `PythonEvaluator` will be used. This default evaluator can parse and interpret Python code in statecharts.

Consider the following example.

```
from sismic.interpreter import Interpreter
from sismic.model import Event

interpreter = Interpreter(my_statechart)
```

The method `execute_once()` returns information about what happened during the execution, including the transitions that were processed, the event that was consumed and the sequences of entered and exited states (see *Macro and micro steps*).

The first call to `execute_once()` puts the statechart in its initial configuration:

```
print('Before:', interpreter.configuration)

step = interpreter.execute_once()

print('After:', interpreter.configuration)
```

```
Before: []
After: ['active', 'floorListener', 'movingElevator', 'doorsOpen', 'floorSelecting']
```

One can send events to the statechart using its `sismic.interpreter.Interpreter.queue()` methods.

```
interpreter.queue(Event('click'))
interpreter.execute_once()  # Process the event
```

For convenience, `queue()` returns `self` and thus can be chained. We will see later that Sismic also provides a way to express scenarios, in order to avoid repeated calls to `queue`.

```
interpreter.queue(Event('click')).queue(Event('click')).execute_once()
```

Notice that `execute_once()` consumes at most one event at a time. In this example, the second *click* event is not processed.

To process all events *at once*, repeatedly call `execute_once()` until it returns a `None` value. For instance:

```
while interpreter.execute_once():
  pass
```

As a shortcut, the `execute()` method will return a list of `sismic.model.MacroStep` instances obtained by repeatedly calling `execute_once()`:

```
from sismic.model import MacroStep

steps = interpreter.execute()
for step in steps:
  assert isinstance(step, MacroStep)
```

Notice that a call to *execute()* first computes the list and **then** returns it, meaning that all the steps are already processed when the call returns.

As a call to *execute()* could lead to an infinite execution (see for example simple/infinite.yaml), an additional parameter `max_steps` can be specified to limit the number of steps that are computed and executed by the method.

```
assert len(interpreter.execute(max_steps=10)) <= 10
```

For convenience, a *Statechart* has an *events_for()* method that returns the list of all possible events that can be interpreted by this statechart (other events will be consumed and ignored). This method also accepts a state name or a list of state names to restrict the list of returned events, and is thus commonly used to get a list of the "interesting" events:

```
print(my_statechart.events_for(interpreter.configuration))
```

Macro and micro steps

An interpreter *execute_once()* (resp. *execute()*) method returns an instance of (resp. a list of) *sismic.model.MacroStep*. A *macro step* corresponds to the process of consuming an event, regardless of the number and the type (eventless or not) of triggered transitions. A macro step also includes every consecutive *stabilization step* (i.e., the steps that are needed to enter nested states, or to switch into the configuration of a history state).

A *MacroStep* exposes the consumed *event* if any, a (possibly empty) list *transitions* of *Transition* instances, and two aggregated ordered sequences of state names, *entered_states* and *exited_states*. In addition, a *MacroStep* exposes a list *sent_events* of events that were fired by the statechart during the considered step. The order of states in those lists determines the order in which their *on entry* and *on exit* actions were processed. As transitions are atomically processed, this means that they could exit a state in *entered_states* that is entered before some state in *exited_states* is exited. The exact order in which states are exited and entered is indirectly available through the *steps* attribute that is a list of all the *MicroStep* that were executed. Each of them contains the states that were exited and entered during its execution, and the a list of events that were sent during the step.

A *micro step* is the smallest, atomic step that a statechart can execute. A *MacroStep* instance thus can be viewed (and is!) an aggregate of *MicroStep* instances.

This way, a complete *run* of a statechart can be summarized as an ordered list of *MacroStep* instances, and details of such a run can be obtained using the *MicroStep* list of a *MacroStep*.

Observing the execution

The interpreter is fully observable during its execution. It provides many public and private attributes that can be used to see what happens. In particular:

- The *execute_once()* (resp. *execute()*) method returns an instance of (resp. a list of) *sismic.model.MacroStep*.
- The `log_trace()` function can be used to log all the steps that were processed during the execution of an interpreter. This methods takes an interpreter and returns a (dynamic) list of macro steps.
- The list of active states can be retrieved using *configuration*.

- The context of the execution is available using `context` (see *Include code in statecharts*).
- It is possible to bind a callable that will be called each time an event is sent by the statechart using the `bind` method of an interpreter (see *Communication between statecharts*).

Include code in statecharts

A statechart can specify code that needs to be executed under some circumstances. For example, the *preamble* property on a statechart, the *guard* or *action* on a transition or the *on entry* and *on exit* properties for a state may all contain code.

In Sismic, these pieces of code can be evaluated and executed by `Evaluator` instances.

Built-in Python code evaluator

By default, Sismic provides two built-in `Evaluator` subclasses:

- A default `PythonEvaluator` that allows the statecharts to execute Python code directly.
- A `DummyEvaluator` that always evaluates to `True` and silently executes nothing when it is called. Its context is an empty dictionary.

The key point to understand how an evaluator works is the concept of `context`, which is a dictionary-like structure that contains the data that is exposed to the code fragments contained in the statechart (ie. override `__locals__`).

As an example, consider the following partial statechart definition.

```
statechart:
  # ...
  preamble: |
    x = 1
    y = 0
  root state:
    name: s1
    on entry: x += 1
```

When the statechart is initialized, the `context` of the `PythonEvaluator` is `{'x': 1, 'y': 0}`. When *s1* is entered, the code will be evaluated with this context. After the execution of `x += 1`, the context associates `2` to `x`.

More precisely, every state and every transition has a specific evaluation context. The code associated with a state is executed in a local context which is composed of local variables and every variable that is defined in the context of the parent state. The context of a transition is built upon the context of its source state.

Note: While you have full access to an ancestor's context, the converse is not true: every variable that is defined in a context is NOT visible by any other context, except the ones that are nested.

When a `PythonEvaluator` instance is initialized, an initial context can be specified:

```
from sismic.code import PythonEvaluator
import math as my_favorite_module

evaluator = PythonEvaluator(initial_context={'x': 1, 'math': my_favorite_module})
```

For convenience, the initial context can be directly provided to the constructor of an `Interpreter`.

Note: The initial context is evaluated *before* any code contained in the statechart. As a consequence, this implies that if a same variable name is used both in the initial context and in the YAML, the value set in the initial context will be overridden by the value set in the YAML definition.

```
yaml = """statechart:
  name: example
  preamble:
    x = 1
  root state:
    name: s
"""

statechart = import_from_yaml(yaml)
interpreter = Interpreter(statechart, initial_context={'x': 2})
print(interpreter.context['x'])
```

In this example, the value of `x` in the statechart is set to `1` while the initial context sets its value to `2`. However, as the initial context is evaluated before the statechart, the value of `x` is `1`.

This is a perfectly normal, expected behavior. If you want to define variables in your statechart that can be overridden by an initial context, you should check this variable does not exist in `locals()`. For example, using

```
if not 'x' in locals():
    x = 1
```

or equivalently,

```
x = locals().get('x', 1)
```

Warning: Under the hood, a Python evaluator makes use of `eval()` and `exec()` with global and local contexts. This can lead to some *weird* issues with variable scope (as in list comprehensions or lambda's). See this question on Stackoverflow for more information.

Features of the built-in Python evaluator

Depending on the situation (state entered, guard evaluation, etc.), the context is populated with additional entries. These entries are covered in the docstring of a *`PythonEvaluator`*:

class `sismic.code.`**`PythonEvaluator`**(*interpreter=None*, ***, *initial_context: typing.Mapping[str, typing.Any] = None*) → None

A code evaluator that understands Python.

Depending on the method that is called, the context can expose additional values:

- **On both code execution and code evaluation:**
  - A *time: float* value that represents the current time exposed by the interpreter.
  - An *active(name: str) -> bool* Boolean function that takes a state name and return *True* if and only if this state is currently active, ie. it is in the active configuration of the `Interpreter` instance that makes use of this evaluator.
- **On code execution:**

- A *send(name: str, **kwargs) -> None* function that takes an event name and additional keyword parameters and raises an internal event with it.
- If the code is related to a transition, the *event: Event* that fires the transition is exposed.

•On guard or contract evaluation:

- If the code is related to a transition, the *event: Event* that fires the transition is exposed.

•On guard or contract (except preconditions) evaluation:

- An *after(sec: float) -> bool* Boolean function that returns *True* if and only if the source state was entered more than *sec* seconds ago. The time is evaluated according to Interpreter's clock.
- A *idle(sec: float) -> bool* Boolean function that returns *True* if and only if the source state did not fire a transition for more than *sec* ago. The time is evaluated according to Interpreter's clock.

•On contract (except preconditions) evaluation:

- A variable *__old__* that has an attribute *x* for every *x* in the context when either the state was entered (if the condition involves a state) or the transition was processed (if the condition involves a transition). The value of *__old__.x* is a shallow copy of *x* at that time.

•On contract evaluation:

- A *sent(name: str) -> bool* function that takes an event name and return True if an event with the same name was sent during the current step.
- A *received(name: str) -> bool* function that takes an event name and return True if an event with the same name is currently processed in this step.

If an exception occurred while executing or evaluating a piece of code, it is propagated by the evaluator.

Each piece of code is executed with (a partially isolated) local context. Every state and every transition has a specific execution context. The code associated with a state is executed in a local context which is composed of local variables and every variable that is defined in the context of the parent state (and so one until the root context is reached). The context of a transition is built upon the context of its source state. The specific context of a state is available through the *context_for* method of a PythonEvaluator.

Parameters

- **`interpreter`** – the interpreter that will use this evaluator, is expected to be an *Interpreter* instance
- **`initial_context`** – a dictionary that will be used as *__locals__*

Note: The documentation below explains how an evaluator is organized and what does the default built-in Python evaluator. Readers that are not interested in tuning existing evaluators or creating new ones can skip this part of the documentation.

Anatomy of a code evaluator

An *Evaluator* must provide two main methods and an attribute:

Evaluator.**_evaluate_code**(*code: str*, ***, *additional_context: typing.Mapping[str, typing.Any] = None*) → bool

Generic method to evaluate a piece of code. This method is a fallback if one of the other evaluate_* methods is not overridden.

Parameters

- **code** – code to evaluate
- **additional_context** – an optional additional context

Returns truth value of *code*

Evaluator.**_execute_code**(*code: str*, ***, *additional_context: typing.Mapping[str, typing.Any] = None*) → typing.List[sismic.model.events.Event]

Generic method to execute a piece of code. This method is a fallback if one of the other execute_* methods is not overridden.

Parameters

- **code** – code to execute
- **additional_context** – an optional additional context

Returns a list of sent events

Evaluator.**context**

The context of this evaluator. A context is a dict-like mapping between variables and values that is expected to be exposed when the code is evaluated.

None of the two methods is actually called by the interpreter during the execution of a statechart. These methods are fallback methods, meaning they are implicitly called when one of the following methods is not defined in a concrete evaluator instance:

class sismic.code.**Evaluator**(*interpreter=None*, ***, *initial_context: typing.Mapping[str, typing.Any] = None*) → None

Abstract base class for any evaluator.

An instance of this class defines what can be done with piece of codes contained in a statechart (condition, action, etc.).

Notice that the execute_* methods are called at each step, even if there is no code to execute. This allows the evaluator to keep track of the states that are entered or exited, and of the transitions that are processed.

Parameters

- **interpreter** – the interpreter that will use this evaluator, is expected to be an *Interpreter* instance
- **initial_context** – an optional dictionary to populate the context

on_step_starts(*event: sismic.model.events.Event = None*) → None

Called each time the interpreter starts a macro step.

Parameters **event** – Optional processed event

execute_statechart(*statechart: sismic.model.statechart.Statechart*) → typing.List[sismic.model.events.Event]

Execute the initial code of a statechart. This method is called at the very beginning of the execution.

Parameters **statechart** – statechart to consider

Returns a list of sent events

evaluate_guard(*transition: sismic.model.elements.Transition*, *event: sismic.model.events.Event*) → bool

Evaluate the guard for given transition.

Parameters

- **transition** – the considered transition
- **event** – instance of *Event* if any

Returns truth value of *code*

execute_action(*transition: sismic.model.elements.Transition*, *event: sismic.model.events.Event*) → typing.List[sismic.model.events.Event]

Execute the action for given transition. This method is called for every transition that is processed, even those with no *action*.

Parameters

- **transition** – the considered transition
- **event** – instance of *Event* if any

Returns a list of sent events

execute_onentry(*state: sismic.model.elements.StateMixin*) → typing.List[sismic.model.events.Event]

Execute the on entry action for given state. This method is called for every state that is entered, even those with no *on_entry*.

Parameters **state** – the considered state

Returns a list of sent events

execute_onexit(*state: sismic.model.elements.StateMixin*) → typing.List[sismic.model.events.Event]

Execute the on exit action for given state. This method is called for every state that is exited, even those with no *on_exit*.

Parameters **state** – the considered state

Returns a list of sent events

evaluate_preconditions(*obj*, *event: sismic.model.events.Event = None*) → typing.Iterable[str]

Evaluate the preconditions for given object (either a *StateMixin* or a *Transition*) and return a list of conditions that are not satisfied.

Parameters

- **obj** – the considered state or transition
- **event** – an optional *Event* instance, in the case of a transition

Returns list of unsatisfied conditions

evaluate_invariants(*obj*, *event: sismic.model.events.Event = None*) → typing.Iterable[str]

Evaluate the invariants for given object (either a *StateMixin* or a *Transition*) and return a list of conditions that are not satisfied.

Parameters

- **obj** – the considered state or transition
- **event** – an optional *Event* instance, in the case of a transition

Returns list of unsatisfied conditions

evaluate_postconditions(*obj*, *event: sismic.model.events.Event = None*) → typing.Iterable[str]

Evaluate the postconditions for given object (either a *StateMixin* or a *Transition*) and return a list of conditions that are not satisfied.

Parameters

- **obj** – the considered state or transition
- **event** – an optional *Event* instance, in the case of a transition

Returns list of unsatisfied conditions

initialize_sequential_conditions(*state: sismic.model.elements.StateMixin*) → None
Initialize sequential conditions.

Parameters **state** – for given state.

update_sequential_conditions(*state: sismic.model.elements.StateMixin*) → typing.Iterable[str]
Update sequential conditions, and return a list of already unsatisfied conditions.

Parameters **state** – for given state

Returns a list of already unsatisfied conditions.

evaluate_sequential_conditions(*state: sismic.model.elements.StateMixin*) → typing.Iterable[str]
Evaluate sequential conditions, and return a list of unsatisfied conditions.

Parameters **state** – for given state

Returns a list of unsatisfied conditions.

In order to understand how the evaluator works, the documentation of the *Evaluator* mentions the following important statements:

- Methods *execute_onentry()* and *execute_onexit()* are called respectively when a state is entered or exited, even if this state does not define a *on_entry* or *on_exit* attribute.
- Method *execute_action()* is called when a transition is processed, even if the transition does not define any *action*.

This allows the evaluator to keep track of the states that are entered or exited, and of the transitions that are processed.

Reproducible scenarios

While events can be sent to an interpreter using its *queue()* method, it can be very convenient to define and store scenarios, ie. sequences or traces of events, that can control the execution of a statechart.

Such scenarios are called *stories* in sismic, and can be used to accurately reproduce the execution of a statechart. They are also very instrumental for testing statecharts.

Writing stories

The module *sismic.stories* provides the building bricks to automate statechart execution. The key concept of this module is the notion of *story*, which is a reproducible sequence of events and pauses that control a statechart interpreter. For our running elevator example, a story may encode things like "*first select the 4th floor, then wait 5 seconds, then select the 2th floor, then wait for another 10 seconds*". This would look as follows:

```
from sismic.stories import Story, Pause, Event

story = Story()
story.append(Event('floorSelected', floor=4))
story.append(Pause(5))
story.append(Event('floorSelected', floor=2))
story.append(Pause(10))
```

For syntactical convenience, the same story can also be written more compactly using a Python iterable:

```
story = Story([Event('floorSelected', floor=4),
               Pause(5),
               Event('floorSelected', floor=2),
               Pause(10)])
```

A instance of *Story* exhibits a *tell()* method that can *tell* the story to an interpreter. Using this method, you can easily reproduce a scenario.

```
# ...  (assume that some interpreter has been created for our elevator statechart)
assert isinstance(interpreter, Interpreter)

story.tell(interpreter)
print(interpreter.time, interpreter.context.get('current'))
```

After having *told* the story to the interpreter, we observe that 15 seconds have passed, and the elevator has moved to the ground floor.

```
15 0
```

While telling a whole story at once can be convenient, it is sometimes interesting to tell the story step by step. The *tell_by_step()* returns a generator that yields the object (either a pause or an event) that was told to the interpreter, and the result of calling *execute()*.

```
# Recreate the interpreter
interpreter = Interpreter(statechart)

for told, macrosteps in story.tell_by_step(interpreter):
    print(interpreter.time, told, interpreter.context.get('current'))
```

```
0 Event('floorSelected', floor=4) 4
5 Pause(5) 4
5 Event('floorSelected', floor=2) 2
15 Pause(10) 0
```

Storywriters

The module *sismic.stories* contains several helper methods to write stories. We expect this module to quickly growth and to provide many ways to automatically generate stories.

Currently, the module contains the following helpers:

sismic.stories.**random_stories_generator**(*items: typing.Sequence[typing.Union[sismic.model.events.Event, sismic.stories.Pause]]*, *length: int = None*, *number: int = None*) → typing.Generator[[sismic.stories.Story, NoneType], NoneType]

A generator that returns random stories whose elements come from *items*. Parameter *items* can be any iterable containing events and/or pauses.

Parameters

- **items** – Items to pick from
- **length** – Length of the story, or *len(items)*
- **number** – number of stories to generate (None = infinite)

Returns An infinite Story generator

sismic.stories.**story_from_trace**(*trace: typing.Iterable[sismic.model.steps.MacroStep]*) → sismic.stories.Story

Return a story that is built upon the given trace (a list of macro steps).

The story is composed of the same pauses and the same events than the ones that generated the given trace. The use case is when you want to reproduce the scenario of an observed behavior.

Notice that internal events are ignored.

Parameters **trace** – A list of *MacroStep* instances.

Returns A story

Design by Contract for statecharts

About Design by Contract

Design by Contract (DbC) was introduced by Bertrand Meyer and popularised through his object-oriented Eiffel programming language. Several other programming languages also provide support for DbC. The main idea is that the specification of a software component (e.g., a method, function or class) is extended with a so-called *contract* that needs to be respected when using this component. Typically, the contract is expressed in terms of preconditions, postconditions and invariants. We have additionally added so-called sequential conditions on top of this.

> Design by contract (DbC), also known as contract programming, programming by contract and design-by-contract programming, is an approach for designing software. It prescribes that software designers should define formal, precise and verifiable interface specifications for software components, which extend the ordinary definition of abstract data types with **preconditions, postconditions and invariants**. These specifications are referred to as "contracts", in accordance with a conceptual metaphor with the conditions and obligations of business contracts. — Wikipedia

DbC for statechart models

While DbC has gained some amount of acceptance at the programming level, there is hardly any support for it at the modeling level.

Sismic aims to change this, by integrating support for Design by Contract for statecharts. The basic idea is that contracts can be defined on statechart componnents (states or transitions), by specifying preconditions, postconditions, invariants and sequential conditions (i.e. conditions that must be sequentially satisfied) on them. At runtime, Sismic will verify the conditions specified by the constracts. If a condition is not satisfied, a *ContractError* will be raised. More specifically, one of the following 4 error types wil be raised: *PreconditionError*, *PostconditionError*, *InvariantError*, or *SequentialConditionError*.

Contracts can be specified for any state contained in the statechart, and for any transition contained in the statechart. A state contract can contain preconditions, postconditions, invariants and/or sequential conditions. At transition level, sequential conditions are not allowed. The semantics for evaluating a contract is as follows:

- **For states:**
  - state preconditions are checked before the state is entered (i.e., **before** executing *on entry*), in the order of occurrence of the preconditions.
  - state postconditions are checked after the state is exited (i.e., **after** executing *on exit*), in the order of occurrence of the postconditions.
  - state invariants are checked at the end of each *macro step*, in the order of occurrence of the invariants. The state must be in the active configuration.

  - sequential conditions on a state are initialized after a state is entered (i.e., **after** executing *on entry*), and evaluated before the state is exited (i.e., **before** executing *on exit*). The evaluation of the sequential condition is updated at each step as long as the state remains in the active configuration.

- **For transitions:**
  - the preconditions are checked before starting the process of the transition (and **before** executing the optional transition action).
  - the postconditions are checked after finishing the process of the transition (and **after** executing the optional transition action).
  - the invariants are checked twice: one before starting and a second time after finishing the process of the transition.

Defining contracts in YAML

Contracts can easily be added to the YAML definition of a statechart (see *Defining statecharts in YAML*) through the use of the *contract* property. Preconditions, postconditions, invariants and sequential conditions are defined as nested items of the *contract* property. The name of these optional contractual conditions is respectively *before* (for preconditions), *after* (for postconditions), *always* (for invariants), and *sequentially* (for sequential conditions):

```
contract:
 - before: ...
 - after: ...
 - always: ...
 - sequentially: ...
```

Obviously, more than one condition of each type can be specified:

```
contract:
 - before: ...
 - before: ...
 - before: ...
 - after: ...
```

A condition is an expression that will be evaluated by an `Evaluator` instance (see *Include code in statecharts*).

```
contract:
 - before: x > 0
 - before: y > 0
 - after: x + y == 0
 - always: x + y >= 0
```

Here is an example of a contracts defined at state level:

```
statechart:
  name: example
  root state:
    name: root
    contract:
     - always: x >= 0
     - always: not active('other state') or x > 0
```

If the default *PythonEvaluator* is used, it is possible to refer to the old value of some variable used in the statechart, by prepending `__old__`. This is particularly useful when specifying postconditions and invariants:

```
contract:
  always: d > __old__.d
  after: (x - __old__.x) < d
```

See the documentation of *PythonEvaluator* for more information.

Sequential conditions

Sequential conditions can be used to describe what should happen when residing in a particular state, and in which order. A sequential condition makes use of some logical and temporal operators, and of *classical* conditions that will be evaluated by an *Evaluator* instance (by default, a *PythonEvaluator* one).

Refer to the documentation of *build_sequence()* for more information about the supported operators. You will never call this function directly, but the documentation explains the implemented mini-language and the supported operators and their semantics.

sismic.code.sequence.**build_sequence**(*expression: str, evaluation_function: typing.Callable[[str], bool] = <built-in function eval>*) → sismic.code.sequence.Sequence

Parse an expression and return the corresponding sequence according to the following mini-language:

- **atom:**
  - "code" or 'code': a fragment of code (e.g. Python code) representing a Boolean expression that evaluates to true or false. The semantics is "satisfied once": as soon as the code evaluates to true once, the truth value of the expression remains true. This is equivalent as "sometimes 'code'" in linear temporal logic.
- **constants:**
  - failure: this constant always evaluates to false.
  - success: this constant always evaluates to true.
- **unary operators:**
  - never A: this expression evaluates to false as soon as expression A evaluates to true.
- **binary operators:**
  - A and B: logical and
  - A or B: logical or
  - A -> B: this is equivalent to "(next always B) since A" in linear temporal logic, i.e. B has to be true (strictly) since A holds. Notice that, due to the "satisfied once" semantics of the atoms, if A and B are atoms, this is merely equivalent to "(A and next (sometimes B))", which means A needs to be true strictly before B or, in other words, A must be satisfied once, then B must be holds once.

Keywords are case-insensitive. Parentheses can be used to group sub expressions. Unary operators have precedence over binary ones (e.g. "A and never B" is equivalent to "A and (never B)"). Unary operators are right associative while binary operators are left associative (e.g. "A and B and C" is equivalent to "(A and B) and C"). The binary operators are listed in decreasing priority (e.g. "A or B and C" is equivalent to "A or (B and C)", and "A and B -> C or D" is equivalent to "(A and B) -> (C or D)").

Examples (assuming that expressions between quotes can be evaluated to true or false):

- "put water" -> "put coffee": ensures water is put before coffee.
- "put water" and "put coffee": ensures water and coffee are put. Due to the "satisfied once" semantics of the atoms, the order in which items are put does not matter.

- (never "put water") or ("put water" -> "put coffee"): if water is put, then coffee must be put too.
- never ("put water" -> "put water"): the condition will fail if water is put twice (but will succeed if water is put once or never put).
- "put water" -> (never "put water"): put water exactly once.

Parameters

- **expression** – an expression to parse
- **evaluation_function** – the function that will be called to evaluate nested pieces of code

Returns a *Sequence* instance.

Please be warned: the syntax allowed in sequential conditions may conflict with YAML's one. Protect your sequential conditions by using quotes or by using the multi-line marker "|".

Example

The following example shows some contract specifications added to the Elevator example.

```
statechart:
  name: Elevator
  preamble: |
    current = 0
    destination = 0
    doors_open = True
  root state:
    name: active
    contract:
      - before: current == 0
      - always: current >= 0  # Floor must be valid
      - always: destination >= 0  # Selected floor must be valid
    parallel states:
      - name: movingElevator
        initial: doorsOpen
        states:
          - name: doorsOpen
            transitions:
              - target: doorsClosed
                guard: destination != current
                action: doors_open = False
              - target: doorsClosed
                guard: after(10) and current > 0
                action: |
                  destination = 0
                  doors_open = False
                contract:
                  - before: current > 0
                  - after: destination == 0
          - name: doorsClosed
            transitions:
              - target: movingUp
                guard: destination > current
              - target: movingDown
                guard: destination < current and destination >= 0
          - name: moving
```

```
            contract:
              - always: not doors_open  # Keep doors closed while moving
              - before: destination != current  # Move only if destination is not␣
↪reached
              - after: destination == current  # Destination should be reached
              - after: current != __old__.current  # Current changed (redundant given␣
↪the two last conditions)
              - sequentially: |
                  (
                    ("destination > current" -> "current > __old__.current")
                    or ("destination < current" -> "current < __old__.current")
                  ) -> "destination == current"
            transitions:
              - target: doorsOpen
                guard: destination == current
                action: doors_open = True
                contract:
                  - before: not doors_open  # Doors are closed
                  - after: doors_open  # Doors are opened
            states:
              - name: movingUp
                on entry: current = current + 1
                contract:
                  - before: current < destination  # Move up only if below
                  - always: current <= destination  # Never go above destination
                  - after: current > __old__.current  # Move up!
                transitions:
                  - target: movingUp
                    guard: destination > current
              - name: movingDown
                contract:
                  - before: current > destination  # Move down only if above
                  - always: current >= destination  # Never go below destination
                  - after: current < __old__.current  # Move down!
                on entry: current = current - 1
                transitions:
                  - target: movingDown
                    guard: destination < current
      - name: floorListener
        initial: floorSelecting
        contract:
          - sequentially: "'received(\"floorSelected\")' -> 'current == destination'␣
↪or ('current != destination' -> 'current == destination')"
        states:
          - name: floorSelecting
            transitions:
              - target: floorSelecting
                event: floorSelected
                action: destination = event.floor
```

Executing statecharts containing contracts

The execution of a statechart that contains contracts does not essentially differ from the execution of a statechart that does not. The only difference is that conditions of each contract are checked at runtime (as explained above) and may raise a subclass of *ContractError*.

```
from sismic.model import Event
from sismic.interpreter import Interpreter
from sismic.io import import_from_yaml

with open('examples/elevator/elevator_contract.yaml') as f:
    statechart = import_from_yaml(f)

    # Make the run fails
    statechart.state_for('movingUp').preconditions[0] = 'current > destination'

    interpreter = Interpreter(statechart)
    interpreter.queue(Event('floorSelected', floor=4))
    interpreter.execute()
```

Here we manually changed one of the preconditions such that it failed at runtime. The exception displays some relevant information to help debug:

```
Traceback (most recent call last):
 ...
sismic.exceptions.PreconditionError: PreconditionError
Object: BasicState('movingUp')
Assertion: current > destination
Configuration: ['active', 'floorListener', 'movingElevator', 'floorSelecting', 'moving
↪']
Step: MicroStep(transition=Transition('doorsClosed', 'movingUp', event=None), entered_
↪states=['moving', 'movingUp'], exited_states=['doorsClosed'])
Context:
 - current = 0
 - destination = 4
 - doors_open = False
```

If you do not want the execution to be interrupted by such exceptions, you can set the `ignore_contract` parameter to `True` when constructing an `Interpreter`. This way, no contract checking will be done during the execution.

Behavior-Driven Development (BDD)

This introduction is inspired by the documentation of Behave, a Python library for Behavior-Driven Development (BDD). BDD is an agile software development technique that encourages collaboration between developers, QA and non-technical or business participants in a software project. It was originally named in 2003 by Dan North as a response to test-driven development (TDD), including acceptance test or customer test driven development practices as found in extreme programming.

BDD focuses on obtaining a clear understanding of desired software behavior through discussion with stakeholders. It extends TDD by writing test cases in a natural language that non-programmers can read. Behavior-driven developers use their native language in combination with the language of domain-driven design to describe the purpose and benefit of their code. This allows developers to focus on why the code should be created, rather than the technical details, and minimizes translation between the technical language in which the code is written and the domain language spoken by the business, users, stakeholders, project management, etc.

The Gherkin language

The Gherkin language is a business readable, domain specific language created to support behavior descriptions in BDD. It lets you describe software's behaviour without the need to know its implementation details. Gherkin allows

the user to describe a software feature or part of a feature by means of representative scenarios of expected outcomes. Like YAML or Python, Gherkin aims to be a human-readable line-oriented language.

Here is an example of a feature and scenario description with Gherkin, describing part of the intended behaviour of the Unix ls command:

```
Feature: ls
In order to see the directory structure
As a UNIX user
I need to be able to list the current directory's contents

Scenario: List 2 files in a directory
    Given I am in a directory "test"
    And I have a file named "foo"
    And I have a file named "bar"
    When I run "ls"
    Then I should get:
        """
        bar
        foo
        """
```

As can be seen above, Gherkin files should be written using natural language - ideally by the non-technical business participants in the software project. Such feature files serve two purposes: documentation and automated tests. Using one of the available Gherkin parsers, it is possible to execute the described scenarios and check the expected outcomes.

See also:

A quite complete overview of the Gherkin language is available here.

BDD and Sismic

Since statecharts are executable pieces of software, it is desirable for statechart users to be able to describe the intended behavior in terms of feature and scenario descriptions. While it is possible to manually integrate the BDD process with any library or software, Sismic is bundled with a command-line utility `sismic-behave` that automates the integration of BDD. `sismic-behave` relies on Behave, a Python library for BDD with full support of the Gherkin language.

As an illustrative example, let us define the desired behavior of our elevator statechart. We first create a feature file that contains several scenarios of interest. By convention, this file has the extension *.feature*, but this is not mandatory. The example illustrates that Sismic provides a set of predefined steps (e.g., *given*, *when*, *then*) to describe common statechart behavior without having to write a single line of Python code.

```
Feature: Elevator

  Scenario: Elevator starts on ground floor
    Then the value of current should be 0
    And the value of destination should be 0

  Scenario: Elevator can move to 7th floor
    When I send event floorSelected with floor=7
    Then the value of current should be 7

  Scenario: Elevator can move to 4th floor
    When I send event floorSelected
      | parameter  | value |
      | floor      | 4     |
      | dummy      | None  |
```

```
    Then the value of current should be 4

  Scenario: Elevator reaches ground floor after 10 seconds
    Given I reproduce "Elevator can move to 7th floor"
    When I wait 10 seconds
    Then the value of current should be 0
    # Notice the variant using "holds":
    And expression current == 0 should hold

  Scenario Outline: Elevator can reach floor from 0 to 5
    When I send event floorSelected with floor=<floor>
    Then the value of current should be <floor>

    Examples:
      | floor |
      | 0     |
      | 1     |
      | 2     |
      | 3     |
      | 4     |
      | 5     |
```

Let us save this file as *elevator.feature* in the same directory as the statechart description, *elevator.yaml*. We can then instruct `sismic-behave` to run on this statechart the scenarios described in the feature file:

```
sismic-behave elevator.yaml --features elevator.feature
```

Note: `sismic-behave` allows to specify the path to each file, so it is not mandatory to put all of them in the same directory. It also accepts multiple files for the `--features` parameter, and supports all the command-line parameters of Behave.

`sismic-behave` will translate the feature file into executable code, compute the outcomes of the scenarios, check whether they match what is expected, and display as summary of all executed scenarios and encountered errors:

```
[...]

1 feature passed, 0 failed, 0 skipped
10 scenarios passed, 0 failed, 0 skipped
22 steps passed, 0 failed, 0 skipped, 0 undefined
Took 0m0.027s
```

Coverage data, including the states that were visited and the transitions that were processed, can be displayed using the `--coverage` argument, as in `sismic-behave statechart.yaml --features tests.feature --coverage`:

```
[...]

State coverage: 55.56%
Entered states: floorSelecting (2) | floorListener (1) | doorsOpen (1) | active (1) |␣
↪movingElevator (1)
Remaining states: doorsClosed | moving | movingDown | movingUp
Transition coverage: 12.50%
Processed transitions: floorSelecting [floorSelected] -> floorSelecting (1)

[...]
```

While BDD can be convenient to identify errors, it is usually not easy to identify where the error occured. `sismic-behave` accepts an additional parameter, namely `--debug-on-error` that drops in a `ipdb` (or the default `pdb` if `ipdb` is not available). The current statechart interpreter is exposed through `context._interpreter` while the current trace is available in `context._steps`. Please consult `pdb`'s documentation for more information on how to use the debugger.

Sismic comes with several predefined steps (see below), but you can easily create your own steps. `sismic-behave` supports a parameter `--steps` which takes a list of Python files containing your own steps. For example, the features in heating_human.feature make use of steps defined in heating_steps.py, and can be tested using `sismic-behave microwave.yaml --features heating_human.feature --steps heating_steps.py` from the *docs/examples/microwave* directory.

All remaining parameters that are provided to `sismic-behave` are passed to `behave`. Notice that `behave` CLI also supports a `--steps` parameter which list the steps that are available). While this `--steps` parameter is overridden by `sismic-behave`, Sismic provides a `--show-steps` parameter that does exactly the same job.

Built-in "Given" and "when" steps

Given/when I do nothing This step should be used to explicitly state that nothing is done.

Given/when I reproduce "{scenario}" Allows to reproduce the steps of given scenario.

```
Scenario: Elevator can move to 7th floor
  When I send event floorSelected with floor=7
  Then the value of current should be 7

Scenario: Elevator reaches ground floor after 10 seconds
  Given I reproduce "Elevator can move to 7th floor"
  When I wait 10 seconds
  Then the value of current should be 0
  # Notice the variant using "holds":
  And expression current == 0 should hold
```

Given/when I repeat step "{step}" {repeat} times Repeat a given step a certain number of times.

Given I disable automatic execution Some steps trigger an automatic execution of the statechart (like sending an event or awaiting). With this step, one can turn of the automatic execution.

Given I enable automatic execution Enable the automatic execution of the statechart (it is enabled by default).

Given/when I import a statechart from {path} Import a statechart from a *yaml* file. This step is implicitly executed when using `sismic-behave`. It is only needed when calling `behave` directly.

Given/when I execute the statechart This step executes the statechart. It is equivalent to *`execute()`*. It should only be used when automatic execution is disabled.

Given/when I execute once the statechart This step executes the statechart once. It is equivalent to *`execute_once()`*. It should only be used when automatic execution is disabled.

Given/when I send event {event_name} This step sends an event to the statechart, and executes the statechart (if automatic execution is enabled).

Given/when I send event {event_name} with {parameter}={value} This step sends an event to the statechart with the given parameter. The value of the parameter is evaluated as Python code. The statechart is executed after this step. Additional parameters can be specified using a table, as follows.

```
Scenario: Elevator can move to 4th floor
  When I send event floorSelected
    | parameter  | value |
```

```
    | floor       | 4     |
    | dummy       | None  |
  Then the value of current should be 4
```

Given/when I wait {seconds} seconds Increment the internal clock of the statechart, and perform a single execution of the statechart. As the execution uses a simulated clock, if the statechart relies on a relative time delta, you should consider using the *repeated* version of this step.

Given/when I wait {seconds} seconds {repeat} times Repeatedly increment the internal clock of the statechart, and execute the statechart after each increment.

Given I set variable {variable} to {value} Set the value of a variable in the internal context of a statechart. The value will be evaluated as Python code.

Built-in "Then" steps

Then state {state_name} should be active Assert that a given state *state_name* is active.

Then state {state_name} should not be active Assert that a given state *state_name* is inactive.

Then event {event_name} should be fired Assert that a given event *event_name* was fired during the last execution.

```
Scenario: Allow heating if door is closed
  Given I send event door_opened
  And I send event item_placed
  And I send event door_closed
  And I send event input_timer_inc
  When I send event input_cooking_start
  Then event heating_on should be fired
```

Then event {event_name} should be fired with {parameter}={value} Assert that a given event *event_name* was fired during the last execution, and that it had an attribute *parameter* with given *value*, evaluated as Python code. Additional parameters can be specified using a table.

Then event {event_name} should not be fired Assert that no event of given name was fired during the last execution.

Then no event should be fired Assert that no event was fired by the statechart during the last execution.

Then variable {variable_name} should be defined Assert that given variable *variable_name* is defined in the context of the statechart.

Then the value of variable {variable_name} should be {value} Assert that given variable *variable_name* is defined and has given *value* in the context of the statechart. The value is evaluated as Python code.

Then expression {expression} should hold Assert that a given *expression* evaluates to true. The expression is evaluated as Python code inside the context of the statechart.

Then the statechart is in a final configuration Assert that the statechart is in a final configuration.

> **Warning:** The list of steps documented hereabove could be incomplete or contain slight variations. An up-to-date list of all the available steps can be found using the `--steps` arguments of `behave`, or using the `--show-steps` argument of `sismic-behave`.

Note: If you do not want to rely on Sismic and want to use *behave* command-line interface, you can easily import the predefined steps using `from sismic.testing.steps import *`. This will also import *behave* and all the

needed objects to define and use new steps. See Python Step Implementations for more information.

Testing statecharts

Like any executable software artefacts, statecharts can and should be tested during their development.

One possible appproach is to test the execution of a statechart *by hand*. The Sismic interpreter stores and returns several values that can be inspected during the execution, including the active configuration, the list of entered or exited states, etc. The functional tests in *tests/test_interpreter.py* on the GitHub repository are several examples of this kind of tests.

This way of testing statecharts is, however, quite cumbersome, especially if one would also like to test a statechart's contracts (i.e., its invariants and behavioral pre- and postconditions).

To overcome this, Sismic provides a module `sismic.testing` that makes it easy to test statecharts using statecharts themselves!

Using statecharts to encode (un)desirable properties

Remark: in the following, the term *statechart under test* refers to the statechart that is to be tested, while the term *property statechart* refers to a statechart that expresses conditions or invariants that should be satisfied by the statechart under test.

While *contracts* can be used to verify assertions on the context of a statechart during its execution, *property statecharts* can be used to test specific behavior of a *statechart under test*.

A *property statechart* defines a property that should (or not) be satisfied by other statecharts. A *property statechart* is like any other statechart, in the sense that neither their syntax nor their semantics differs from any other statechart. The difference comes from the events it receives and the role it plays. If the run of a *statechart property* ends in a final state, it signifies that the property was verified. In the case of a *desirable* property, this means that the test succeed. In the case of an *undesirable* property, this means that the test failed.

Note: This is more a convention than a requirement, but you should follow it.

The run of such a *property statechart* is driven by a specific sequence of events and pauses, which represents what happens during the execution of a *statechart under test*.

For example, such a sequence contains *event consumed* events, *state entered* events, *state exited* events, ... In particular, the following events are generated:

- A *execution started* event is sent at the beginning.
- each time a step begins, a *step started* event is created.
- each time an event is consumed, a *event consumed* event is created. the consumed event is available through the *event* attribute.
- each time a state is exited, an *state exited* event is created. the name of the state is available through the *state* attribute.
- each time a transition is processed, a *transition processed* event is created. the source state name and the target state name (if any) are available respectively through the *source* and *target* attributes. The event processed by the transition is available through the *event* attribute.

- each time a state is entered, an *state entered* event is created. the name of the state is available through the *state* attribute.
- each time a step ends, a *step ended* event is created.
- A *execution stopped* event is sent at the end.
- each time an event is fired from within the statechart, a *sent event* is created. the sent event is available through the *event* attribute.

The sequence does follow the interpretation order:

1. an event is possibly consumed
2. For each matching transition
   (a) states are exited
   (b) transition is processed
   (c) states are entered
   (d) internal events are sent
   (e) statechart is stabilized (some states are exited and/or entered, some events are sent)

Using statechart to check properties on a trace

The trace of an interpreter is the list of its executed macro steps. The trace can be built upon the values returned by each call to *execute()* (or *execute_once()*), or can be automatically built using `sismic.interpreter.helpers.log_trace()` function.

Function *teststory_from_trace()* provides an easy way to construct a story for statechart properties from the trace obtained by executing a *statechart under test*.

`sismic.testing.`**`teststory_from_trace`**(*trace: typing.List[sismic.model.steps.MacroStep]*) → sismic.stories.Story

Return a test story based on the given *trace*, a list of macro steps. See documentation to see which are the events that are generated.

Notice that this function adds a *pause* if there is any delay between pairs of consecutive steps.

Parameters **`trace`** – a list of *MacroStep* instances

Returns A story

Notice that using this function, the property statechart can not access the context of the statechart under test.

To summarize, if you want to test the **trace** of a *statechart under test* `tested`, you need to:

1. construct a *property statechart* `tester` that expresses the property you want to test.
2. execute `tested` (using a story or directly by sending events) and log its trace.
3. generate a new story from this trace with *teststory_from_trace()*.
4. tell this story to an interpreter of the *property statechart* `tester`.

If `tester` ends in a final configuration, ie. `tester.final` holds, then the test is **considered** successful. The semantic of *successful* depends on the *desirability* of the checked property.

The following *property statechart* examples are relative to *this statechart*. They show the specification of some testers in YAML, and how to execute them.

Note that these statechart properties are currently used as unit tests for Sismic.

7th floor is never reached

This *property statechart* ensures that the 7th floor is never reached. It stores the current floor based on the number of times the elevator goes up and goes down.

```
statechart:
  name: Test that the elevator never reachs 7th floor
  preamble: floor = 0
  root state:
    name: root
    initial: standing
    states:
      - name: standing
        transitions:
          - event: state entered
            guard: event.state == 'moving'
            target: moving
          - guard: floor == 7
            target: fail
      - name: moving
        transitions:
          - event: state entered
            guard: event.state == 'movingUp'
            action: floor += 1
          - event: state entered
            guard: event.state == 'movingDown'
            action: floor -= 1
          - event: state exited
            guard: event.state == 'moving'
            target: standing
      - name: fail
        type: final
```

It can be tested as follows:

```
    def test_7th_floor_never_reached(self):
        story = Story([Event('floorSelected', floor=8)])
        trace = story.tell(self.tested)  # self.tested is an interpreter for our␣
↪elevator

        test_story = teststory_from_trace(trace)

        with open('docs/examples/elevator/tester_elevator_7th_floor_never_reached.yaml
↪') as f:
            tester = Interpreter(io.import_from_yaml(f))
        test_story.tell(tester)
        self.assertFalse(tester.final)
```

You can even simulate a failure:

```
    def test_7th_floor_never_reached_fails(self):
        story = Story([Event('floorSelected', floor=4), Pause(2), Event('floorSelected
↪', floor=7)])
        trace = story.tell(self.tested)  # self.tested is an interpreter for our␣
↪elevator

        test_story = teststory_from_trace(trace)
```

```
        with open('docs/examples/elevator/tester_elevator_7th_floor_never_reached.yaml
↪') as f:
            tester = Interpreter(io.import_from_yaml(f))
        test_story.tell(tester)
        self.assertTrue(tester.final)
```

Elevator moves after 10 seconds

This *property statechart* checks that the elevator automatically moves after some idle time if it is not on the ground floor. The test sets a timeout of 12 seconds, but it should work for any number strictly greater than 10 seconds.

```
statechart:
  name: Test that the elevator goes to ground floor after 10 seconds (timeout set to␣
↪12 seconds)
  preamble: floor = 0
  root state:
    name: root
    initial: active
    states:
      - name: active
        parallel states:
          - name: guess floor
            transitions:
              - event: state entered
                guard: event.state == 'movingUp'
                action: floor += 1
              - event: state entered
                guard: event.state == 'movingDown'
                action: floor -= 1
          - name: check timeout
            initial: standing
            states:
              - name: standing
                transitions:
                  - event: state entered
                    guard: event.state == 'moving'
                    target: moving
                  - guard: after(12) and floor != 0
                    target: timeout
              - name: moving
                transitions:
                  - event: state exited
                    guard: event.state == 'moving'
                    target: standing
              - name: timeout
                type: final
```

We check this tester using several stories, as follows:

```
    def test_elevator_moves_after_10s(self):
        stories = [
            Story([Event('floorSelected', floor=4)]),
            Story([Event('floorSelected', floor=0)]),
            Story([Event('floorSelected', floor=4), Pause(10)]),
            Story([Event('floorSelected', floor=0), Pause(10)]),
            Story([Event('floorSelected', floor=4), Pause(9)]),
```

```
            Story([Event('floorSelected', floor=0), Pause(9)]),
        ]

        for story in stories:
            with self.subTest(story=story):
                # Reopen because we need to reset it
                with open('docs/examples/elevator/elevator.yaml') as f:
                    sc = io.import_from_yaml(f)
                tested = Interpreter(sc)

                test_story = teststory_from_trace(story.tell(tested))

                with open('docs/examples/elevator/tester_elevator_moves_after_10s.yaml
↪') as f:
                    tester = Interpreter(io.import_from_yaml(f))
                test_story.tell(tester)
                self.assertFalse(tester.final)
```

Using statecharts to check properties at runtime

Sismic provides a convenience class to allow *property statechart* to check properties at runtime. Class *ExecutionWatcher* can be used to associate a statechart tester with a *statechart under test*:

class sismic.testing.**ExecutionWatcher**(*tested_interpreter: sismic.interpreter.interpreter.Interpreter*) → None

This can be used to associate a property statechart with a statechart under test. An instance of this class is built upon an *Interpreter* instance (the tested one).

It provides a method, namely *watch_with* which takes a property statechart (and a set of optional parameters that can be used to tune the interpreter that will be built upon this property statechart) and returns the resulting *Interpreter* instance for this tester.

If started (using *start*), whenever something happens during the execution of the interpreter under test, events are automatically sent to every associated statechart properties. Their internal clock are synchronized, and the context of the statechart under test is also exposed to the property statechart, ie. if *x* is a variable in the context of a statechart under test, then *context.x* is dynamically exposed to every associated property statechart.

Parameters **tested_interpreter** – Interpreter to watch

start() → None

Send a *started* event to the statechart properties, and starts watching the execution of the statechart under test.

stop() → None

Send a *stopped* event to the statechart properties, and stops watching the execution of the statechart under test.

watch_with(*property_statechart: sismic.model.statechart.Statechart*, *fails_fast: bool = False*, *interpreter_klass: typing.Callable[..., sismic.interpreter.interpreter.Interpreter] = <class 'sismic.interpreter.interpreter.Interpreter'>*, ***kwargs*) → sismic.interpreter.interpreter.Interpreter

Watch the execution of the tested interpreter with given sproperty statechart.

interpreter_klass is a callable that accepts a *Statechart* instance, an *initial_context* parameter and any additional parameters provided to this method. This callable must return an *Interpreter* instance

Parameters

- **property_statechart** – a property statechart (instance of *Statechart*)

- **fails_fast** – If True (default is False), the execution of the statechart under test will raise an AssertionError as soon as given property statechart reaches a final state.
- **interpreter_klass** – a callable that accepts a *Statechart* instance, an *initial_context* and any additional (optional) parameters provided to this method.

Returns the interpreter instance that wraps given property statechart.

To summarize, if you want to test (**at runtime**) the execution of a *statechart under test* `tested`, you need to:

1. create an *`ExecutionWatcher`* with `tested`.
2. construct at least one *property statechart* `tester` that expresses the property you want to test.
3. associate each `tester` to the watcher with *`watch_with()`*.
4. start watching with *`start()`*.
5. execute `tested` (using a story or directly by sending events).
6. stop watching with *`stop()`*.

If `tester` ends in a final configuration, ie. `tester.final` holds, then the test is **considered** successful. Again, the semantic of a *successful* run depends on the *desirability* of the property.

Destination should be reached

This *property statechart* ensures that every chosen destination is finally reached.

```
statechart:
  name: Test that destinations are reached
  preamble: |
    destinations = []  # List of destinations
  root state:
    name: root
    initial: check
    states:
      - name: check
        transitions:
          - event: execution stopped
            guard: len(destinations) > 0
            target: fail
        parallel states:
          - name: wait floor selection
            transitions:
              - event: event consumed
                guard: event.event.name == 'floorSelected'
                action: destinations.append(event.event.floor)
          - name: wait doors open
            transitions:
              - event: state entered
                guard: event.state == 'doorsOpen'
                action: |
                  # Current floor, deduced from tested statechart's context
                  floor = context.current

                  # Remove floor from destination if it exists
                  try:
                    destinations.remove(floor)
                  except ValueError:
                    pass
```

```
      - name: fail
        type: final
```

It can be tested as follows:

```
from sismic.io import import_from_yaml
from sismic.interpreter import Interpreter
from sismic.testing import ExecutionWatcher
from sismic.model import Event

# Load statecharts
with open('examples/elevator/elevator.yaml') as f:
    elevator_statechart = import_from_yaml(f)
with open('examples/elevator/tester_elevator_destination_reached.yaml') as f:
    tester_statechart = import_from_yaml(f)

# Create the interpreter and the watcher
interpreter = Interpreter(elevator_statechart)
watcher = ExecutionWatcher(interpreter)

# Add the tester and start watching
tester = watcher.watch_with(tester_statechart)
watcher.start()

# Send the elevator to 4th
interpreter.queue(Event('floorSelected', floor=4)).execute(max_steps=2)
assert tester.context['destinations'] == [4]

interpreter.execute()
assert tester.context['destinations'] == []

# Stop watching. The statechart ends in a final state only if a failure occured
watcher.stop()

assert not tester.final
```

Dealing with time

It is quite usual in a statechart to rely on some notion of time. To cope with this, the built-in evaluator (see *PythonEvaluator*) has support for time events `after(x)` and `idle(x)`, meaning that a transition can be triggered after a certain amount of time.

When it comes to interpreting statecharts, Sismic deals with time using an internal clock whose value is exposed by the *time* property of an *Interpreter*. Basically, this clock does nothing by itself except for being available for an *Evaluator* instance. If your statechart needs to rely on a time value, you have to set it by yourself.

Below are some examples to illustrate the use of time events.

Simulated time

Sismic provides a discrete step-by-step interpreter for statecharts. It seems natural in a discrete simulation to rely on simulated time.

The following example illustrates a statechart modeling the behavior of a simple *elevator*. If the elevator is sent to the 4th floor, according to the YAML definition of this statechart, the elevator should automatically go back to the ground

floor after 10 seconds.

```
- target: doorsClosed
  guard: after(10) and current > 0
  action: destination = 0
```

Rather than waiting for 10 seconds, one can simulate this. First, one should load the statechart and initialize the interpreter:

```
from sismic.io import import_from_yaml
from sismic.interpreter import Interpreter
from sismic.model import Event

with open('examples/elevator/elevator.yaml') as f:
    statechart = import_from_yaml(f)

interpreter = Interpreter(statechart)
```

The internal clock of our interpreter is `0`. This is, `interpreter.time == 0` holds. We now ask our elevator to go to the 4th floor.

```
interpreter.queue(Event('floorSelected', floor=4))
interpreter.execute()
```

The elevator should now be on the 4th floor. We inform the interpreter that 2 seconds have elapsed:

```
interpreter.time += 2
print(interpreter.execute())
```

The output should be an empty list `[]`. Of course, nothing happened since the condition `after(10)` is not satisfied yet. We now inform the interpreter that 8 additional seconds have elapsed.

```
interpreter.time += 8
print(interpreter.execute())
```

The output now contains a list of steps, from which we can see that the elevator has moved down to the ground floor. We can check the current floor:

```
print(interpreter.context.get('current'))
```

This displays `0`.

Real time

If a statechart needs to be aware of a real clock, the simplest way to achieve this is by using the `time.time()` function of Python. In a nutshell, the idea is to synchronize `interpreter.time` with a real clock. Let us first initialize an interpreter using one of our statechart example, the *elevator*:

```
from sismic.io import import_from_yaml
from sismic.interpreter import Interpreter
from sismic.model import Event

with open('examples/elevator/elevator.yaml') as f:
    statechart = import_from_yaml(f)

interpreter = Interpreter(statechart)
```

The interpreter initially sets its clock to 0. As we are interested in a real-time simulation of the statechart, we need to set the internal clock of our interpreter. We import from `time` a real clock, and store its value into a `starttime` variable.

```
import time
starttime = time.time()
```

We can now execute the statechart by sending a `floorSelected` event, and wait for the output. For our example, we first ask the statechart to send to elevator to the 4th floor.

```
interpreter.queue(Event('floorSelected', floor=4))
interpreter.execute()
print('Current floor:', interpreter.context.get('current'))
print('Current time:', interpreter.time)
```

At this point, the elevator is on the 4th floor and is waiting for another input event. The internal clock value is still 0.

```
Current floor: 4
Current time: 0
```

We should inform our interpreter of the new current time. Of course, as our interpreter follows a discrete simulation, nothing really happens until we call *execute()* or *execute_once()*.

```
interpreter.time = time.time() - starttime
# Does nothing if (time.time() - starttime) is less than 10!
interpreter.execute()
```

Assuming you quickly wrote these lines of code, nothing happened. But if you wait a little bit, and update the clock again, it should move the elevator to the ground floor.

```
interpreter.time = time.time() - starttime
interpreter.execute()
```

And *voilà*!

As it is not very convenient to manually set the clock each time you want to execute something, it is best to put it in a loop. To avoid the use of a `starttime` variable, you can set the initial time of an interpreter using the `initial_time` parameter of its constructor. This is illustrated in the following example.

```
from sismic.io import import_from_yaml
from sismic.interpreter import Interpreter
from sismic.model import Event

import time

# Load statechart and create an interpreter
with open('examples/elevator.yaml') as f:
    statechart = import_from_yaml(f)

# Set the initial time
interpreter = Interpreter(statechart)
interpreter.time = time.time()

# Send an initial event
interpreter.queue(Event('floorSelected', floor=4))

while not interpreter.final:
    interpreter.time = time.time()
```

```
    if interpreter.execute():
        print('something happened at time {}'.format(interpreter.time))

    time.sleep(0.5)  # 500ms
```

Here, we called the `sleep()` function to slow down the loop (optional). The output should look like:

```
something happened at time 1450383083.9943285
something happened at time 1450383093.9920669
```

As our statechart does not define any way to reach a final configuration, the `not interpreter.final` condition always holds, and the execution needs to be interrupted manually.

Asynchronous execution

Notice from previous example that using a loop makes it impossible to send events to the interpreter. For convenience, sismic provides a `sismic.interpreter.helpers.run_in_background()` function that run an interpreter in a thread, and does the job of synchronizing the clock for you.

Note: An optional argument `callback` can be passed to `run_in_background()`. It must be a callable that accepts the (possibly empty) list of *`MacroStep`* returned by the underlying call to *`execute()`*.

Communication between statecharts

It is not unusual to have to deal with multiple distinct components in which the behavior of a component is driven by things that happen in the other components. One can model such a situation using a single statechart with parallel states, or by plugging several statecharts into one main statechart (see *`sismic.model.Statechart.copy_from_statechart()`*). The communication and synchronization between the components can be done either by using `active(state_name)` in guards, or by sending internal events that address other components.

However, we believe that this approach is not very convenient:

- all the components must be defined in a single statechart;
- state name collision could occur;
- components must share a single execution context;
- component composition is not easy to achieve
- ...

Sismic allows to define multiple components in multiple statecharts, and brings a way for those statecharts to communicate and synchronize via events.

Binding statecharts

Every instance of *`Interpreter`* exposes a *`bind()`* method which allows to bind statecharts.

`Interpreter.`**`bind`**(*interpreter_or_callable: typing.Union[typing.Interpreter, typing.Callable[[sismic.model.events.Event], typing.Any]]*) → sismic.interpreter.interpreter.Interpreter

Bind an interpreter or a callable to the current interpreter. Each time an internal event is sent by this interpreter,

any bound object will be called with the same event. If *interpreter_or_callable* is an *Interpreter* instance, its *queue* method is called. This is, if *i1* and *i2* are interpreters, *i1.bind(i2)* is equivalent to *i1.bind(i2.queue)*.

Parameters **interpreter_or_callable** – interpreter or callable to bind

Returns *self* so it can be chained

When an interpreter `interpreter_1` is bound to an interpreter `interpreter_2` using `interpreter_1.bind(interpreter_2)`, the **internal** events that are sent by `interpreter_1` are automatically propagated as **external** events to `interpreter_2`. The binding is not restricted to only two statecharts. For example, assume we have three instances of *Interpreter*:

```
assert isinstance(interpreter_1, Interpreter)
assert isinstance(interpreter_2, Interpreter)
assert isinstance(interpreter_3, Interpreter)
```

We define a bidirectional communication between the two first interpreters:

```
interpreter_1.bind(interpreter_2)
interpreter_2.bind(interpreter_1)
```

We also bind the third interpreters with the two first ones. Notice that *bind()* returns the current interpreter, so multiple calls can be chained:

```
interpreter_3.bind(interpreter_1).bind(interpreter_2)
```

When an internal event is sent by an interpreter, the bound interpreters also receive this event as an external event. In the last example, when an internal event is sent by `interpreter_3`, then a corresponding external event is sent both to `interpreter_1` and `interpreter_2`.

Note: Practically, unless you subclassed *Interpreter*, the only difference between internal and external events are the priority order in which they are processed by the interpreter.

```
from sismic.model import InternalEvent, Event

# Manually create and raise an internal event
interpreter_3.raise_event(InternalEvent('test'))

print('Events for interpreter_1:', interpreter_1._external_events.pop())
print('Events for interpreter_2:', interpreter_2._external_events.pop())
print('Events for interpreter_3:', interpreter_3._internal_events.pop())
```

```
Events for interpreter_1: Event('test')
Events for interpreter_2: Event('test')
Events for interpreter_3: InternalEvent('test')
```

Example

Consider our running example, the elevator statechart. This statechart expects to receive *floorSelected* events (with a *floor* parameter representing the selected floor). The statechart operates autonomously, provided that we send such events.

Let us define a new statechart that models a panel of buttons for our elevator. For example, we consider that our panel has 4 buttons numbered 0 to 3.

```
statechart:
  name: Elevator buttons
  description: |
    Buttons that remotely control the elevator.
  root state:
    name: active
    parallel states:
      - name: button_0
        transitions:
          - event: button_0_pushed
            action: send('floorSelected', floor= 0)
      - name: button_1
        transitions:
          - event: button_1_pushed
            action: send('floorSelected', floor= 1)
      - name: button_2
        transitions:
          - event: button_2_pushed
            action: send('floorSelected', floor= 2)
      - name: button_3
        transitions:
          - event: button_3_pushed
            action: send('floorSelected', floor= 3)
```

As you can see in the YAML version of this statechart, the panel expects an event for each button: *button_0_pushed*, *button_1_pushed*, *button_2_pushed* and *button_3_pushed*. Each of those event causes the execution of a transition which, in turn, creates and sends a *floorSelected* event. The *floor* parameter of this event corresponds to the button number.

We bind our panel with our elevator, such that the panel can control the elevator:

```
from sismic.io import import_from_yaml
from sismic.interpreter import Interpreter
from sismic.model import Event, InternalEvent

elevator = Interpreter(import_from_yaml(open('examples/elevator/elevator.yaml')))
buttons = Interpreter(import_from_yaml(open('examples/elevator/elevator_buttons.yaml
↪')))

# Elevator will receive events from buttons
buttons.bind(elevator)
```

Events that are sent **to** `buttons` are not propagated, but events that are sent **by** `buttons` are automatically propagated to `elevator`:

```
print('Awaiting events in buttons:', list(buttons._external_events))  # Empty
buttons.queue(Event('button_2_pushed'))

print('Awaiting events in buttons:', list(buttons._external_events))  # External event

buttons.execute(max_steps=2)  # (1) initialize buttons, and (2) consume button_2_
↪pushed
print('Awaiting events in buttons:', list(buttons._internal_events))
print('Awaiting events in elevator:', list(elevator._external_events))
```

```
Awaiting events in buttons: []
Awaiting events in buttons: [Event('button_2_pushed')]
```

```
Awaiting events in buttons: [InternalEvent('floorSelected', floor=2)]
Awaiting events in elevator: [Event('floorSelected', floor=2)]
```

The execution of bound statecharts does not differ from the execution of unbound statecharts:

```
elevator.execute()
print('Current floor:', elevator.context.get('current'))
```

```
Current floor: 2
```

Integrate statecharts into your code

Sismic provides several ways to integrate executable statecharts into your Python source code. The simplest way is to directly *embed* the entire code in the statechart's description. This was illustrated with the Elevator example in *Include code in statecharts*. Its code is part of the YAML file of the statechart, and interpreted by Sismic during the statechart's execution.

In order to make a statechart communicate with the source code contained in the environment in which it is executed, there are basically two approaches:

1. The statechart sends events to, or receives external events from the environment.
2. The environment stores shared objects in the statechart's initial context, and the statechart calls operations on these objects and/or accesses the variables contained in it.

Of course, one could also use a hybrid approach, combining ideas from the three approches above.

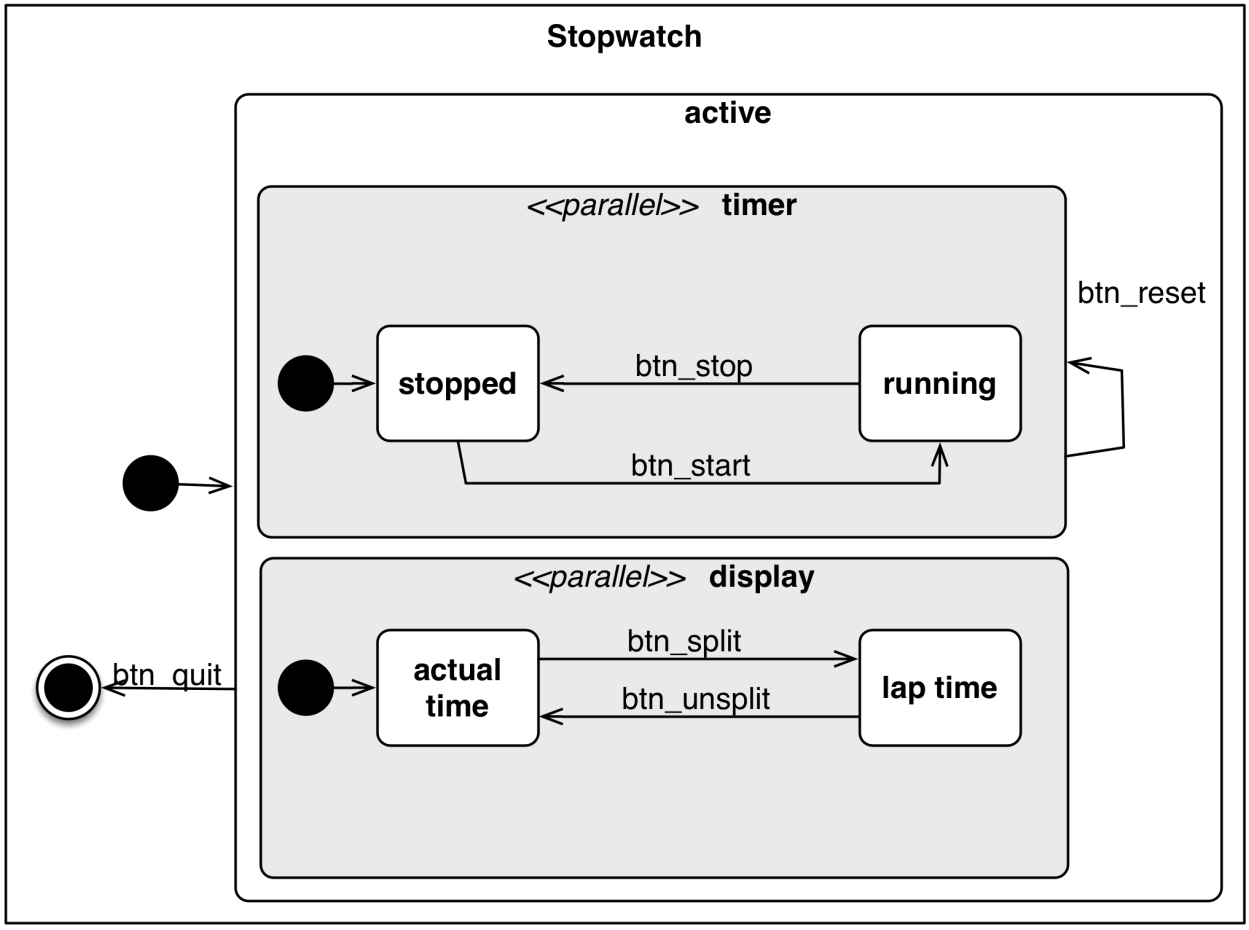
Running example

In this document, we will present the main differences between the two approaches, on the basis of a simple example of a Graphical User Interface (GUI) whose behaviour is defined by a statechart. All the source code and YAML files for this example, discussed in more detail below, is available in the *docs/examples* directory of Sismic's repository.

The example represents a simple stopwatch, i.e., a timer than can be started, stopped and reset. It also provides a split time feature and a display of the elapsed time. A button-controlled GUI of such a stopwatch looks as follows (inactive buttons are greyed out):

Essentially, the stopwatch simply displays a value, representing the elapsed time (expressed in seconds), which is initially 0. By clicking on the *start* button the stopwatch starts running. When clicking on *stop*, the stopwatch stops running. By using *split*, the time being displayed is temporarily frozen, although the stopwatch continues to run. Clicking on *unsplit* while continue to display the actual elapsed time. *reset* will restart from 0, and *quit* will quit the stopwatch application.

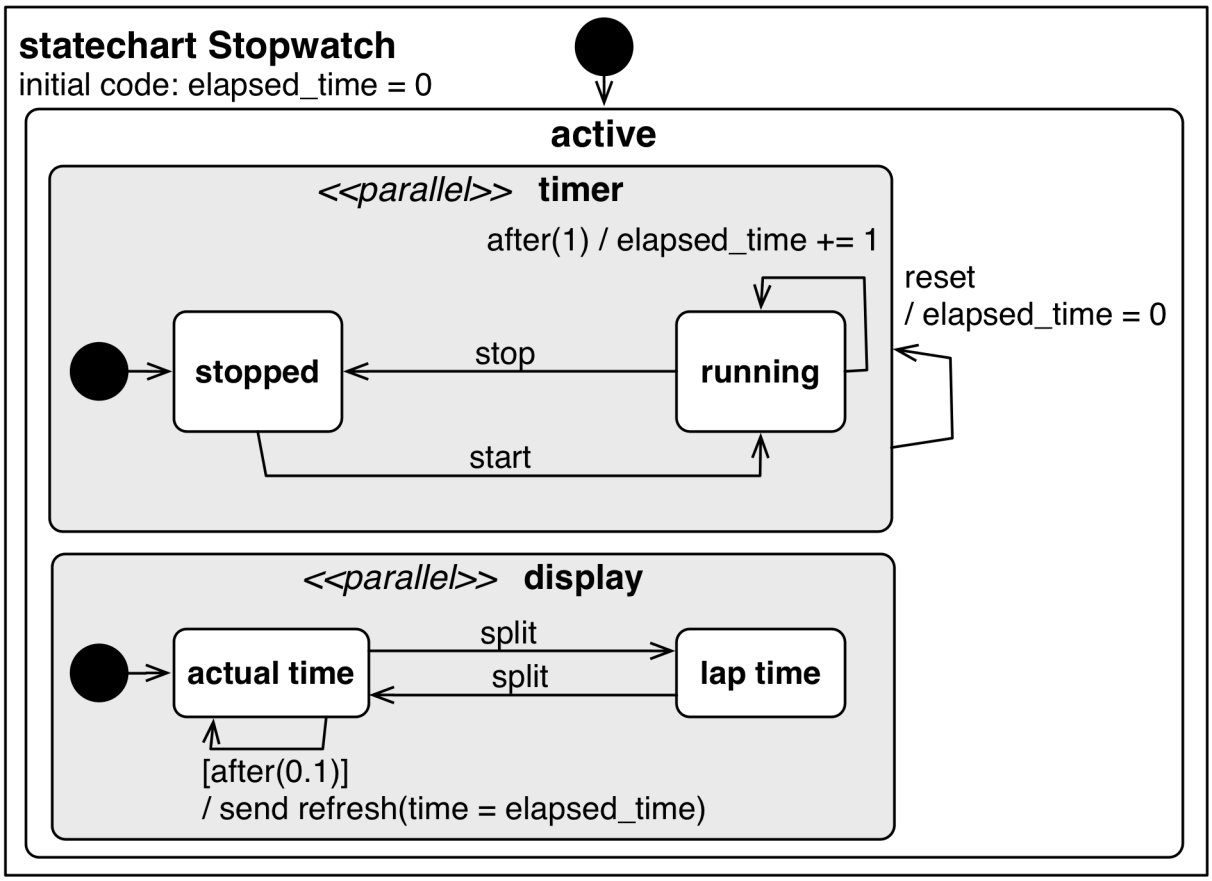
The idea is that the buttons will trigger state changes and actions carried out by an underlying statechart. Taking abstraction of the concrete implementation, the statechart would essentially look as follows, with one main *active* state containing two parallel substates *timer* and *display*.



Controlling a statechart from within the environment

Let us illustrate how to control a statechart through source code that executes in the environment containing the statechart. The statechart's behaviour is triggered by external events sent to it by the source code each time one of the buttons in the GUI is pressed. Conversely, the statechart itself can send events back to the source code to update its display.

This statechart looks as follows:



Here is the YAML file containing the textual description of this statechart:

```
statechart:
  name: Stopwatch
  description: |
    A simple stopwatch which support "start", "stop", "split", and "reset".
    These features are triggered respectively using "start", "stop", "split", and
↪"reset".

    The stopwatch sends an "refresh" event each time the display is updated.
    The value to display is attached to the event under the key "time".

    The statechart is composed of two parallel regions:
     - A "timer" region which increments "elapsed_time" if timer is running
     - A "display" region that refreshes the display according to the actual time/lap␣
↪time feature

  preamble: elapsed_time = 0
  root state:
    name: active
    parallel states:
      - name: timer
        initial: stopped
        transitions:
          - event: reset
            action: elapsed_time = 0
        states:
          - name: running
```

```
            transitions:
              - event: stop
                target: stopped
              - guard: after(1)
                target: running
                action: elapsed_time += 1
          - name: stopped
            transitions:
              - event: start
                target: running
      - name: display
        initial: actual time
        states:
          - name: actual time
            transitions:
              - guard: after(0.2)
                target: actual time
                action: |
                  send('refresh', time=elapsed_time)
              - event: split
                target: lap time
          - name: lap time
            transitions:
              - event: split
                target: actual time
```

We observe that the statechart contains an `elapsed_time` variable, that is updated every second while the stopwatch is in the *running* state. The statechart will modify its behaviour by receiving *start*, *stop*, *reset* and *split* events from its external environment. In parallel to this, every 100 milliseconds, the *display* state of the statechart sends a *refresh* event (parameterised by the `time` variable containing the `elapsed_time` value) back to its external environment. In the *lap time* state (reached through a *split* event) , this regular refreshing is stopped until a new *split* event is received.

The source code (shown below) that defines the GUI of the stopwatch, and that controls the statechart by sending it events, is implemented using the `Tkinter` library. Each button of the GUI is bound to a Python method in which the corresponding event is created and sent to the statechart. The statechart is *bound* to the source code by defining a new *`Interpreter`* that contains the parsed YAML specification, and using the *`bind()`* method. The `event_handler` passed to it allows the Python source code to receive events back from the statechart. In particular, the `w_timer` field of the GUI will be updated with a new value of the time whenever the statechart sends a *refresh* event. The `run` method, which is put in Tk's mainloop, updates the internal clock of the interpreter and executes the interpreter.

```
import time
import tkinter as tk

from sismic.interpreter import Interpreter
from sismic.io import import_from_yaml
from sismic.model import Event


# The two following lines are NOT needed in a typical environment.
# These lines make sismic available in our testing environment
import sys
sys.path.append('../../..')
```

```
# Create a tiny GUI
class StopwatchApplication(tk.Frame):
    def __init__(self, master=None):
        super().__init__(master)

        # Initialize widgets
        self.create_widgets()

        # Create a Stopwatch interpreter
        with open('stopwatch.yaml') as f:
            statechart = import_from_yaml(f)
        self.interpreter = Interpreter(statechart)
        self.interpreter.time = time.time()

        # Bind interpreter events to the GUI
        self.interpreter.bind(self.event_handler)

        # Run the interpreter
        self.run()

    def run(self):
        # This function does essentially the same job than ``sismic.interpreter.run_
↪in_background``
        # but uses Tkinter's mainloop instead of a Thread, which is more adequate.

        # Update internal clock and execute interpreter
        self.interpreter.time = time.time()
        self.interpreter.execute()

        # Queue a call every 100ms on tk's mainloop
        self.after(100, self.run)

        # Update the widget that contains the list of active states.
        self.w_states['text'] = 'active states: ' + ', '.join(self.interpreter.
↪configuration)

    def create_widgets(self):
        self.pack()

        # Add buttons
        self.w_btn_start = tk.Button(self, text='start', command=self._start)
        self.w_btn_stop = tk.Button(self, text='stop', command=self._stop)
        self.w_btn_split = tk.Button(self, text='split', command=self._split)
        self.w_btn_unsplit = tk.Button(self, text='unsplit', command=self._unsplit)
        self.w_btn_reset = tk.Button(self, text='reset', command=self._reset)
        self.w_btn_quit = tk.Button(self, text='quit', command=self._quit)

        # Initial button states
        self.w_btn_stop['state'] = tk.DISABLED
        self.w_btn_unsplit['state'] = tk.DISABLED

        # Pack
        self.w_btn_start.pack(side=tk.LEFT,)
        self.w_btn_stop.pack(side=tk.LEFT,)
        self.w_btn_split.pack(side=tk.LEFT,)
        self.w_btn_unsplit.pack(side=tk.LEFT,)
        self.w_btn_reset.pack(side=tk.LEFT,)
        self.w_btn_quit.pack(side=tk.LEFT,)
```

```
        # Active states label
        self.w_states = tk.Label(root)
        self.w_states.pack(side=tk.BOTTOM, fill=tk.X)

        # Timer label
        self.w_timer = tk.Label(root, font=("Helvetica", 16), pady=5)
        self.w_timer.pack(side=tk.BOTTOM, fill=tk.X)

    def event_handler(self, event):
        # Update text widget when timer value is updated
        if event.name == 'refresh':
            self.w_timer['text'] = event.time

    def _start(self):
        self.interpreter.queue(Event('start'))
        self.w_btn_start['state'] = tk.DISABLED
        self.w_btn_stop['state'] = tk.NORMAL

    def _stop(self):
        self.interpreter.queue(Event('stop'))
        self.w_btn_start['state'] = tk.NORMAL
        self.w_btn_stop['state'] = tk.DISABLED

    def _reset(self):
        self.interpreter.queue(Event('reset'))

    def _split(self):
        self.interpreter.queue(Event('split'))
        self.w_btn_split['state'] = tk.DISABLED
        self.w_btn_unsplit['state'] = tk.NORMAL

    def _unsplit(self):
        self.interpreter.queue(Event('split'))
        self.w_btn_split['state'] = tk.NORMAL
        self.w_btn_unsplit['state'] = tk.DISABLED

    def _quit(self):
        self.master.destroy()


if __name__ == '__main__':
    # Create GUI
    root = tk.Tk()
    root.wm_title('StopWatch')
    app = StopwatchApplication(master=root)

    app.mainloop()
```

Controlling the environment from within the statechart

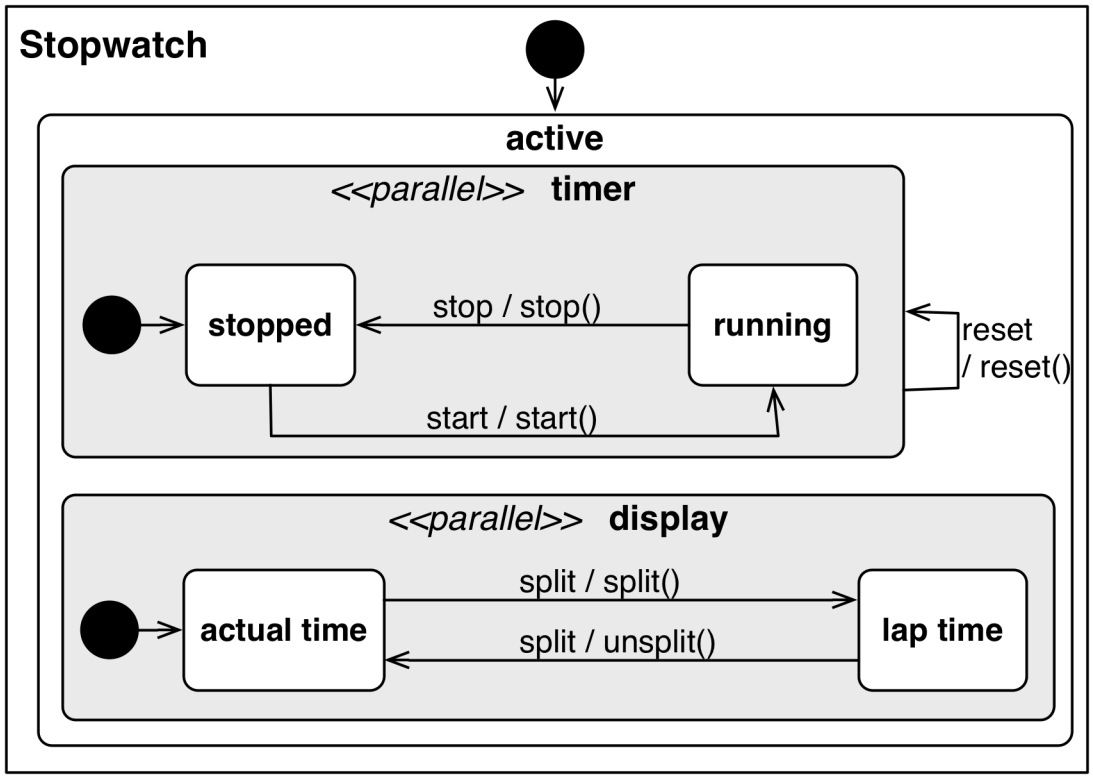
In this second example, we basically reverse the idea: now the Python code that resides in the environment contains the logic (e.g., the `elapsed_time` variable), and this code is exposed to, and controlled by, a statechart that represents the main loop of the program and calls the necessary methods in the source code. These method calls are associated to actions on the statechart's transitions. With this solution, the statechart is no longer a *black box*, since it needs to be aware of the source code, in particular the methods it needs to call in this code.

An example of the Python code that is controlled by the statechart is given below:

```
class Stopwatch:
    def __init__(self):
        self.elapsed_time = 0
        self.split_time = 0
        self.is_split = False
        self.running = False

    def start(self):
        # Start internal timer
        self.running = True

    def stop(self):
        # Stop internal timer
        self.running = False

    def reset(self):
        # Reset internal timer
        self.elapsed_time = 0

    def split(self):
        # Split time
        if not self.is_split:
            self.is_split = True
            self.split_time = self.elapsed_time

    def unsplit(self):
        # Unsplit time
        if self.is_split:
            self.is_split = False

    def display(self):
        # Return the value to display
        if self.is_split:
            return int(self.split_time)
        else:
            return int(self.elapsed_time)

    def update(self, delta):
        # Update internal timer of ``delta`` seconds
        if self.running:
            self.elapsed_time += delta
```

The statechart expects such a `Stopwatch` instance to be created and provided in its initial context. Recall that an *Interpreter* accepts an optional `initial_context` parameter. In this example, `initial_context={'stopwatch': Stopwatch()}`.

The statechart is simpler than in the previous example: one parallel region handles the running status of the stopwatch, and a second one handles its split features.




```
statechart:
  name: Stopwatch
  description: |
    A simple stopwatch which support "start", "stop", "split", and "reset".
    These features are triggered respectively using "start", "stop", "split", and
↪"reset".

    The stopwatch expects a "stopwatch" object in its initial context.
    This object should support the following methods: "start", "stop", "split", "reset
↪", and "unsplit".
  root state:
    name: active
    parallel states:
      - name: timer
        initial: stopped
        transitions:
          - event: reset
            action: stopwatch.reset()
        states:
          - name: running
            transitions:
              - event: stop
                target: stopped
                action: stopwatch.stop()
          - name: stopped
            transitions:
              - event: start
                target: running
                action: stopwatch.start()
      - name: display
        initial: actual time
```

```
        states:
          - name: actual time
            transitions:
              - event: split
                target: lap time
                action: stopwatch.split()
          - name: lap time
            transitions:
              - event: split
                target: actual time
                action: stopwatch.unsplit()
```

The Python code of the GUI no longer needs to *listen* to the events sent by the interpreter. It should, of course, continue to send events (corresponding to button presses) to the statechart using `send`. The *binding* between the statechart and the GUI is now achieved differently, by simply passing the `stopwatch` object to the *`Interpreter`* as its `initial_context`.

```
import time
import tkinter as tk

from sismic.interpreter import Interpreter
from sismic.io import import_from_yaml
from sismic.model import Event
from stopwatch import Stopwatch


# The two following lines are NOT needed in a typical environment.
# These lines make sismic available in our testing environment
import sys
sys.path.append('../../..')




# Create a tiny GUI
class StopwatchApplication(tk.Frame):
    def __init__(self, master=None):
        super().__init__(master)

        # Initialize widgets
        self.create_widgets()

        # Create a Stopwatch interpreter
        with open('stopwatch_external.yaml') as f:
            statechart = import_from_yaml(f)

        # Create a stopwatch object and pass it to the interpreter
        self.stopwatch = Stopwatch()
        self.interpreter = Interpreter(statechart, initial_context={'stopwatch': self.
↪stopwatch})
        self.interpreter.time = time.time()

        # Run the interpreter
        self.run()

        # Update the stopwatch every 100ms
        self.after(100, self.update_stopwatch)
```

```
    def update_stopwatch(self):
        self.stopwatch.update(delta=0.1)
        self.after(100, self.update_stopwatch)

        # Update timer label
        self.w_timer['text'] = self.stopwatch.display()

    def run(self):
        # Queue a call every 100ms on tk's mainloop
        self.interpreter.execute()
        self.after(100, self.run)
        self.w_states['text'] = 'active states: ' + ', '.join(self.interpreter.
↪configuration)

    def create_widgets(self):
        self.pack()

        # Add buttons
        self.w_btn_start = tk.Button(self, text='start', command=self._start)
        self.w_btn_stop = tk.Button(self, text='stop', command=self._stop)
        self.w_btn_split = tk.Button(self, text='split', command=self._split)
        self.w_btn_unsplit = tk.Button(self, text='unsplit', command=self._unsplit)
        self.w_btn_reset = tk.Button(self, text='reset', command=self._reset)
        self.w_btn_quit = tk.Button(self, text='quit', command=self._quit)

        # Initial button states
        self.w_btn_stop['state'] = tk.DISABLED
        self.w_btn_unsplit['state'] = tk.DISABLED

        # Pack
        self.w_btn_start.pack(side=tk.LEFT,)
        self.w_btn_stop.pack(side=tk.LEFT,)
        self.w_btn_split.pack(side=tk.LEFT,)
        self.w_btn_unsplit.pack(side=tk.LEFT,)
        self.w_btn_reset.pack(side=tk.LEFT,)
        self.w_btn_quit.pack(side=tk.LEFT,)

        # Active states label
        self.w_states = tk.Label(root)
        self.w_states.pack(side=tk.BOTTOM, fill=tk.X)

        # Timer label
        self.w_timer = tk.Label(root, font=("Helvetica", 16), pady=5)
        self.w_timer.pack(side=tk.BOTTOM, fill=tk.X)

    def _start(self):
        self.interpreter.queue(Event('start'))
        self.w_btn_start['state'] = tk.DISABLED
        self.w_btn_stop['state'] = tk.NORMAL

    def _stop(self):
        self.interpreter.queue(Event('stop'))
        self.w_btn_start['state'] = tk.NORMAL
        self.w_btn_stop['state'] = tk.DISABLED

    def _reset(self):
        self.interpreter.queue(Event('reset'))
```

```
    def _split(self):
        self.interpreter.queue(Event('split'))
        self.w_btn_split['state'] = tk.DISABLED
        self.w_btn_unsplit['state'] = tk.NORMAL

    def _unsplit(self):
        self.interpreter.queue(Event('split'))
        self.w_btn_split['state'] = tk.NORMAL
        self.w_btn_unsplit['state'] = tk.DISABLED

    def _quit(self):
        self.master.destroy()


if __name__ == '__main__':
    # Create GUI
    root = tk.Tk()
    root.wm_title('StopWatch (external)')
    app = StopwatchApplication(master=root)

    app.mainloop()
```

Implementing other statechart semantics

An *Interpreter* makes use of several *private* methods for its initialization and computations. These methods computes the transition(s) that should be processed, the resulting steps, etc. These methods can be overridden or combined easily to define other variants of the statechart semantics.

Interpreter.**_select_event**() → typing.Union[sismic.model.events.Event, NoneType]

Return (and consume!) the next available event if any. This method prioritizes internal events over external ones.

Returns An instance of Event or None if no event is available

Interpreter.**_select_transitions**(*event: sismic.model.events.Event = None*) → typing.List[sismic.model.elements.Transition]

Return a list of transitions that can be triggered according to the given event, or eventless transition if *event* is None.

Parameters **event** – event to consider

Returns a list of *Transition* instances

Interpreter.**_filter_transitions**(*transitions: typing.List[sismic.model.elements.Transition]*) → typing.List[sismic.model.elements.Transition]

Given a list of transitions, return a filtered list of transitions with respect to the inner-first/source-state semantic.

Parameters **transitions** – a list of *Transition* instances

Returns a list of *Transition* instances

Interpreter.**_sort_transitions**(*transitions: typing.List[sismic.model.elements.Transition]*) → typing.List[sismic.model.elements.Transition]

Given a list of triggered transitions, return a list of transitions in an order that represents the order in which they have to be processed.

Parameters **transitions** – a list of *Transition* instances

Returns an ordered list of *Transition* instances

Raises ***ExecutionError*** – In case of non-determinism (*NonDeterminismError*) or conflicting transitions (*ConflictingTransitionsError*).

`Interpreter._create_steps`(*event: sismic.model.events.Event, transitions: typing.Iterable[sismic.model.elements.Transition]*) → typing.List[sismic.model.steps.MicroStep]

Return a (possibly empty) list of micro steps. Each micro step corresponds to the process of a transition matching given event.

Parameters

- **event** – the event to consider, if any
- **transitions** – the transitions that should be processed

Returns a list of micro steps.

`Interpreter._create_stabilization_step`(*names: typing.Iterable[str]*) → sismic.model.steps.MicroStep

Return a stabilization step, ie. a step that lead to a more stable situation for the current statechart. Stabilization means:

•Enter the initial state of a compound state with no active child

•Enter the memory of a history state

•Enter the children of an orthogonal state with no active child

•Exit active states if all "deepest" (leaves) states are final

Parameters **names** – List of states to consider (usually, the active configuration)

Returns A *MicroStep* instance or *None* if this statechart can not be more stabilized

`Interpreter._apply_step`(*step: sismic.model.steps.MicroStep*) → sismic.model.steps.MicroStep

Apply given *MicroStep* on this statechart

Parameters **step** – *MicroStep* instance

Returns a new MicroStep, completed with sent events

These methods are called directly (or not) by *execute_once*.

See also:

Consider looking at the source of *execute_once* to understand how these methods are related and organized.

Example: Outer-first/source-state semantics

For example, in order to obtain an outer-first/source-state semantics (instead of the inner-first/source-state one that Sismic provides by default), one should subclass *Interpreter* and override *_filter_transitions*.

Example: Semantics where internal events have no priority

If you want to change the semantics of Sismic so that internal events no longer have priority over external events, it suffices to override the *_select_event()* method and to invert the order in which the internal and external events queues are visited.

Example: Custom way to deal with non-determinism

If you want to change the way the Sismic semantics deals with non-determinism, for example because it deviates from the semantics given by SCXML or Rhapsody (remember *Statechart semantics*), you can implement your own variant for dealing with non-determinism. The method `_sort_transitions()` is where the whole job is done:

1. It looks for non-determinism in (non-parallel) transitions,
2. It looks for conflicting transitions in parallel transitions,
3. It sorts the kept transitions based on our semantic.

According to your needs, adapt the content of this method.

Credits

Development Lead

- Alexandre Decan

Contributors

- Tom Mens
- Mathieu Goeminne
- Ali Parsai

Changelog

0.22.11 (2017-01-12)

- (Fixed) Path error when using `sismic-behave` on Windows.

0.22.10 (2016-11-25)

- (Added) A `--debug-on-error` parameter for `sismic-behave`.

0.22.9 (2016-11-25)

- (Fixed) Behave step "Event x should be fired" now checks that the event was fired during the last execution.

0.22.8 (2016-10-19)

- (Fixed) YAML values like "1", "1.0", "yes", "True" are converted to strings, not to int, float and bool respectively.
- (Changed) `ruamel.yaml` replaces `pyyaml` as supported YAML parser.
- (Changed) Use `schema` instead of `pykwalify` (which unfortunately freezes its dependencies versions) to validate (the structure of) YAML files.

- (Changed) `import_from_yaml` raises `StatechartError` instead of `SchemaError` if it cannot validate given YAML against the predefined schema.

0.22.7 (2016-08-19)

- (Added) A new helper `coverage_from_trace` that returns coverage information (in absolute numbers) from a trace.
- (Added) Parameter `fails_fast` (default is `False`, behavior preserved) for `ExecutionWatcher.watch_with` methods. This parameter allows the watcher to raise an `AssertionError` as soon as the added watcher reaches a final configuration.
- (Changed) `StateMixin`, `Transition` and `Event`'s `__eq__` method returns a `NotImplemented` object if the other object involved in the comparison is not an instance of the same class, meaning that `Event('a') == 1` now raises a `NotImplementedError` instead of being `False`.

0.22.6 (2016-08-03)

- (Changed) `Event`, `MacroStep`, `MicroStep`, `StateMixin`, `Transition`, `Statechart` and `Interpreter`'s `__repr__` returns a valid Python expression.
- (Changed) The context returned by a `PythonEvaluator` (and thus by the default `Interpreter`) exhibits nested variables (the ones that are not defined in the preamble of a statechart). Those variables are prefixed by the name of the state in which they are declared, to avoid name clashing.
- (Changed) Context variables are sorted in exceptions' `.__str__` methods.

0.22.4 (2016-07-08)

- (Added) `sismic-behave` CLI now accepts a `--steps` parameter, which is a list of file paths containing the steps implementation.
- (Added) `sismic-behave` CLI now accepts a `--show-steps` parameter, which list the steps (equivalent to Behave's overriden `--steps` parameter).
- (Added) `sismic-behave` now returns an appropriate exit code.
- (Changed) Reorganisation of `docs/examples`.
- (Fixed) Coverage data for `sismic-behave` takes the initialization step into account (regression introduced in 0.21.0).

0.22.3 (2016-07-06)

- (Added) `sent` and `received` are also available in preconditions and postconditions.

0.22.2 (2016-07-01)

- (Added) `model.Event` is now correctly pickled, meaning that Sismic can be used in a multiprocessing environment.

0.22.1 (2016-06-29)

- (Added) A *event {event_name} should not be fired* steps for BDD.
- (Added) Both `MicroStep` and `MacroStep` have a list `sent_events` of events that were sent during the step.
- (Added) Property statecharts receive a `event sent` event when an event is sent by the statechart under test.
- (Changed) Events fired from within the statechart are now collected and sent at the end of the current micro step, instead of being immediately sent.
- (Changed) Invariants and sequential contracts are now evaluated ordered by their state's depth

0.22.0 (2016-06-13)

- (Added) Support for sequential conditions in contracts (see documentation for more information).
- (Added) Python code evaluator: *after* and *idle* are now available in postconditions and invariants.
- (Added) Python code evaluator: *received* and *sent* are available in invariants.
- (Added) An `Evaluator` has now a `on_step_starts` method which is called at the beginning of each step, with the current event (if any) being processed.
- (Added) `Interpreter.raise_event` to send events from within the statechart.
- (Added) A `copy_from_statechart` method for a `Statechart` instance that allows to copy (part of) a statechart into a state.
- (Added) Microwave controller example (see *docs/examples/microwave.[yaml|py]*).
- (Changed) Events sent by a code evaluator are now returned by the `execute_*` methods instead of being automatically added to the interpreter's queue.
- (Changed) Moved `run_in_background` and `log_trace` from `sismic.interpreter` to the newly added `sismic.interpreter.helpers`.
- (Changed) Internal API changes: rename `self.__x` to `self._x` to avoid (mostly) useless name mangling.

0.21.0 (2016-04-22)

Changes for `interpreter.Interpreter` class:

- (Removed) `_select_eventless_transition` which is a special case of `_select_transition`.
- (Added) `_select_event`, extracted from `execute_once`.
- (Added) `_filter_transitions`, extracted from `_select_transition`.
- (Changed) `_execute_step` is now `_apply_step`.
- (Changed) `_compute_stabilization_step` is now `_create_stabilization_step` and accepts a list of state names
- (Changed) `_compute_transitions_step` is now `_create_steps`.
- (Changed) Except for the `statechart` parameter, all the parameters for `Interpreter`'s constructor can now be only provided by name.
- (Fixed) Contracts on a transition are checked (if not explicitly disabled) even if the transition has no *action*.
- (Fixed) `Evaluator.execute_action` is called even if the transition has no *action*.

- (Fixed) States are added/removed from the active configuration as soon as they are entered/exited. Previously, the configuration was only updated at the end of the step (and could possibly lead to inaccurate results when using `active(name)` in a `PythonEvaluator`).

The default `PythonEvaluator` class has been completely rewritten:

- (Changed) Code contained in states and/or transitions is now executed with a local context instead of a global one. The local context of a state is built upon the local context of its parent, and so one until the local context of the statechart is reached. This should facilitate the use of dummy variables in nested states and transitions.
- (Changed) The code is now compiled (once) before is evaluation/execution. This should increase performance.
- (Changed) The frozen context of a state (ie. `__old__`) is now computed only if contracts are checked, and only if at least one invariant or one postcondition exists.
- (Changed) The `initial_context` parameter of `Evaluator`'s constructor can now only be provided by name.
- (Changed) The `additional_context` parameter of `Evaluator._evaluate_code` and `Evaluator._execute_code` can now only be provided by name.

Miscellaneous:

- (Fixed) Step *I load the statechart* now executes (once) the statechart in order to put it into a stable initial configuration (regression introduced in 0.20.0).

0.20.5 (2016-04-14)

- (Added) Type hinting (see PEP484 and mypy-lang project)

0.20.4 (2016-03-25)

- (Changed) Statechart testers are now called property statechart.
- (Changed) Property statechart can describe *desirable* and *undesirable* properties.

0.20.3 (2016-03-22)

- (Changed) Step *Event x should be fired* now checks sent events from the beginning of the test, not only for the last executed step.
- (Fixed) Internal events that are sequentially sent are now sequentially consumed (and not anymore in reverse order).

0.20.2 (2016-02-24)

- (Fixed) `interpreter.log_trace` does not anymore log empty macro step.

0.20.1 (2016-02-19)

- (Added) A *step ended* event at the end of each step in a tester story.
- (Changed) The name of the events and attributes that are exposed in a tester story has changed. Consult the documentation for more information.

0.20.0 (2016-02-17)

- (Added) Module `interpreter` provides a `log_trace` function that takes an interpreter instance and returns a (dynamic) list of executed macro steps.
- (Added) Module `testing` exposes an `ExecutionWatcher` class that can be used to check statechart properties with tester statecharts at runtime.
- (Changed) `Interpreter.__init__` does not anymore stabilize the statechart. Stabilization is done during the first call of `execute_once`.
- (Changed) `Story.tell` returns a list of `MacroStep` (the *trace*) instead of an `Interpreter` instance.
- (Changed) The name of some attributes of an event in a tester story changes (e.g. *event* becomes *consumed_event*, *state* becomes *entered_state* or *exited_state* or *source_state* or *target_state*).
- (Removed) `Interpreter.trace`, as it can be easily obtained from `execute_once` or using `log_trace`.
- (Removed) `Interpreter.__init__` does not accept an `initial_time` parameter.
- (Fixed) Parallel state without children does not any more result into an infinite loop.

0.19.0 (2016-02-10)

- (Added) BDD can now output coverage data using `--coverage` command-line argument.
- (Changed) The YAML definition of a statechart must use *root state:* instead of *initial state:*.
- (Changed) When a contract is evaluated by a `PythonEvaluator`, `__old__.x` raises an `AttributeError` instead of a `KeyError` if `x` does not exist.
- (Changed) Behave is now called from Python instead of using a subprocess and thus allows debugging.

0.18.1 (2016-02-03)

- (Added) Support for behavior-driven-development using Behave.

0.17.3 (2016-01-29)

- (Added) An `io.text.export_to_tree` that returns a textual representation of the states.
- (Changed) `Statechart.rename_to` does not anymore raise `KeyError` but `exceptions.StatechartError`.
- (Changed) Wheel build should work on Windows

0.17.1 (2016-01-25)

Many backward incompatible changes in this update, especially if you used to work with `model`. The YAML format of a statechart also changed, look carefully at the changelog and the documentation.

- (Added) YAML: an history state can declare *on entry* and *on exit*.
- (Added) Statechart: new methods to manipulate transitions: `transitions_from`, `transitions_to`, `transitions_with`, `remove_transition` and `rotate_transition`.

- (Added) Statechart: new methods to manipulate states: `remove_state`, `rename_state`, `move_state`, `state_for`, `parent_for`, `children_for`.
- (Added) Steps: `__eq__` for `MacroStep` and `MicroStep`.
- (Added) Stories: `tell_by_step` method for a `Story`.
- (Added) Testing: `teststory_from_trace` generates a *step* event at the beginning of each step.
- (Added) Module: a new exceptions hierarchy (see `exceptions` module). The new exceptions are used in place of the old ones (`Warning`, `AssertionError` and `ValueError`).
- (Changed) YAML: uppermost *states:* should be replaced by *initial state:* and can contain at most one state.
- (Changed) YAML: uppermost *on entry:* should be replaced by *preamble:*.
- (Changed) YAML: initial memory of an history state should be specified using *memory* instead of *initial*.
- (Changed) YAML: contracts for a statechart must be declared on its root state.
- (Changed) Statechart: rename `StateChart` to `Statechart`.
- (Changed) Statechart: rename `events` to `events_for`.
- (Changed) Statechart: `states` attribute is now `Statechart.state_for` method.
- (Changed) Statechart: `register_state` is now `add_state`.
- (Changed) Statechart: `register_transition` is now `add_transition`.
- (Changed) Statechart: now defines a root state.
- (Changed) Statechart: checks done in `validate`.
- (Changed) Transition: `.event` is a string instead of an `Event` instance.
- (Changed) Transition: attributes `from_state` and `to_state` are renamed into `source` and `target`.
- (Changed) Event: `__eq__` takes `data` attribute into account.
- (Changed) Event: `event.foo` raises an `AttributeError` instead of a `KeyError` if `foo` is not defined.
- (Changed) State: `StateMixin.name` is now read-only (use `Statechart.rename_state`).
- (Changed) State: split `HistoryState` into a mixin `HistoryStateMixin` and two concrete subclasses, namely `ShallowHistoryState` and `DeepHistoryState`.
- (Changed) IO: Complete rewrite of `io.import_from_yaml` to load states before transitions. Parameter names have changed.
- (Changed) Module: adapt module hierarchy (no visible API change).
- (Changed) Module: expose module content through `__all__`.
- (Removed) Transition: `transitions` attribute on `TransitionStateMixin`, use `Statechart.transitions_for` instead.
- (Removed) State: `CompositeStateMixin.children`, use `Statechart.children_for` instead.

0.16.0 (2016-01-15)

- (Added) An `InternalEvent` subclass for `model.Event`.
- (Added) `Interpreter` now exposes its `statechart`.
- (Added) `Statechart.validate` checks that a targeted compound state declares an initial state.

- (Changed) `Interpreter.queue` does not accept anymore an `internal` parameter. Use an instance of `InternalEvent` instead (#20).
- (Fixed) `Story.story_from_trace` now ignores internal events (#19).
- (Fixed) Condition C3 in `Statechart.validate`.

0.15.0 (2016-01-12)

- (Changed) Rename `Interpreter.send` to `Interpreter.queue` (#18).
- (Changed) Rename `evaluator` module to `code`.

0.14.3 (2016-01-12)

- (Added) Changelog.
- (Fixed) Missing files in MANIFEST.in

API Reference

Module *code*

class `sismic.code.`**`Evaluator`**(*interpreter=None*, ***, *initial_context: typing.Mapping[str, typing.Any] = None*) → None

Bases: `object`

Abstract base class for any evaluator.

An instance of this class defines what can be done with piece of codes contained in a statechart (condition, action, etc.).

Notice that the execute_* methods are called at each step, even if there is no code to execute. This allows the evaluator to keep track of the states that are entered or exited, and of the transitions that are processed.

Parameters

- **`interpreter`** – the interpreter that will use this evaluator, is expected to be an *Interpreter* instance
- **`initial_context`** – an optional dictionary to populate the context

`context`

The context of this evaluator. A context is a dict-like mapping between variables and values that is expected to be exposed when the code is evaluated.

`evaluate_guard`(*transition: sismic.model.elements.Transition*, *event: sismic.model.events.Event*) → bool

Evaluate the guard for given transition.

Parameters

- **`transition`** – the considered transition
- **`event`** – instance of *Event* if any

Returns truth value of *code*

evaluate_invariants(*obj*, *event: sismic.model.events.Event = None*) → typing.Iterable[str]
Evaluate the invariants for given object (either a *StateMixin* or a *Transition*) and return a list of conditions that are not satisfied.

Parameters

- **obj** – the considered state or transition
- **event** – an optional *Event* instance, in the case of a transition

Returns list of unsatisfied conditions

evaluate_postconditions(*obj*, *event: sismic.model.events.Event = None*) → typing.Iterable[str]
Evaluate the postconditions for given object (either a *StateMixin* or a *Transition*) and return a list of conditions that are not satisfied.

Parameters

- **obj** – the considered state or transition
- **event** – an optional *Event* instance, in the case of a transition

Returns list of unsatisfied conditions

evaluate_preconditions(*obj*, *event: sismic.model.events.Event = None*) → typing.Iterable[str]
Evaluate the preconditions for given object (either a *StateMixin* or a *Transition*) and return a list of conditions that are not satisfied.

Parameters

- **obj** – the considered state or transition
- **event** – an optional *Event* instance, in the case of a transition

Returns list of unsatisfied conditions

evaluate_sequential_conditions(*state: sismic.model.elements.StateMixin*) → typing.Iterable[str]
Evaluate sequential conditions, and return a list of unsatisfied conditions.

Parameters **state** – for given state

Returns a list of unsatisfied conditions.

execute_action(*transition: sismic.model.elements.Transition*, *event: sismic.model.events.Event*) → typing.List[sismic.model.events.Event]
Execute the action for given transition. This method is called for every transition that is processed, even those with no *action*.

Parameters

- **transition** – the considered transition
- **event** – instance of *Event* if any

Returns a list of sent events

execute_onentry(*state: sismic.model.elements.StateMixin*) → typing.List[sismic.model.events.Event]
Execute the on entry action for given state. This method is called for every state that is entered, even those with no *on_entry*.

Parameters **state** – the considered state

Returns a list of sent events

execute_onexit(*state: sismic.model.elements.StateMixin*) → typing.List[sismic.model.events.Event]

Execute the on exit action for given state. This method is called for every state that is exited, even those with no *on_exit*.

Parameters **state** – the considered state

Returns a list of sent events

execute_statechart(*statechart: sismic.model.statechart.Statechart*) → typing.List[sismic.model.events.Event]

Execute the initial code of a statechart. This method is called at the very beginning of the execution.

Parameters **statechart** – statechart to consider

Returns a list of sent events

initialize_sequential_conditions(*state: sismic.model.elements.StateMixin*) → None

Initialize sequential conditions.

Parameters **state** – for given state.

on_step_starts(*event: sismic.model.events.Event = None*) → None

Called each time the interpreter starts a macro step.

Parameters **event** – Optional processed event

update_sequential_conditions(*state: sismic.model.elements.StateMixin*) → typing.Iterable[str]

Update sequential conditions, and return a list of already unsatisfied conditions.

Parameters **state** – for given state

Returns a list of already unsatisfied conditions.

class sismic.code.**DummyEvaluator**(*interpreter=None*, *, *initial_context=None*)

Bases: sismic.code.evaluator.Evaluator

A dummy evaluator that does nothing and evaluates every condition to True.

evaluate_guard(*transition: sismic.model.elements.Transition*, *event: sismic.model.events.Event*) → bool

Evaluate the guard for given transition.

Parameters

- **transition** – the considered transition
- **event** – instance of *Event* if any

Returns truth value of *code*

evaluate_invariants(*obj*, *event: sismic.model.events.Event = None*) → typing.Iterable[str]

Evaluate the invariants for given object (either a *StateMixin* or a *Transition*) and return a list of conditions that are not satisfied.

Parameters

- **obj** – the considered state or transition
- **event** – an optional *Event* instance, in the case of a transition

Returns list of unsatisfied conditions

evaluate_postconditions(*obj*, *event: sismic.model.events.Event = None*) → typing.Iterable[str]

Evaluate the postconditions for given object (either a *StateMixin* or a *Transition*) and return a list of conditions that are not satisfied.

Parameters

- **obj** – the considered state or transition
- **event** – an optional *Event* instance, in the case of a transition

Returns list of unsatisfied conditions

evaluate_preconditions(*obj*, *event: sismic.model.events.Event = None*) → typing.Iterable[str]

Evaluate the preconditions for given object (either a *StateMixin* or a *Transition*) and return a list of conditions that are not satisfied.

Parameters

- **obj** – the considered state or transition
- **event** – an optional *Event* instance, in the case of a transition

Returns list of unsatisfied conditions

evaluate_sequential_conditions(*state: sismic.model.elements.StateMixin*) → typing.Iterable[str]

Evaluate sequential conditions, and return a list of unsatisfied conditions.

Parameters **state** – for given state

Returns a list of unsatisfied conditions.

execute_action(*transition: sismic.model.elements.Transition*, *event: sismic.model.events.Event*) → typing.List[sismic.model.events.Event]

Execute the action for given transition. This method is called for every transition that is processed, even those with no *action*.

Parameters

- **transition** – the considered transition
- **event** – instance of *Event* if any

Returns a list of sent events

execute_onentry(*state: sismic.model.elements.StateMixin*) → typing.List[sismic.model.events.Event]

Execute the on entry action for given state. This method is called for every state that is entered, even those with no *on_entry*.

Parameters **state** – the considered state

Returns a list of sent events

execute_onexit(*state: sismic.model.elements.StateMixin*) → typing.List[sismic.model.events.Event]

Execute the on exit action for given state. This method is called for every state that is exited, even those with no *on_exit*.

Parameters **state** – the considered state

Returns a list of sent events

execute_statechart(*statechart: sismic.model.statechart.Statechart*) → typing.List[sismic.model.events.Event]

Execute the initial code of a statechart. This method is called at the very beginning of the execution.

Parameters **statechart** – statechart to consider

Returns a list of sent events

initialize_sequential_conditions(*state: sismic.model.elements.StateMixin*) → None
Initialize sequential conditions.

Parameters **state** – for given state.

on_step_starts(*event: sismic.model.events.Event = None*) → None
Called each time the interpreter starts a macro step.

Parameters **event** – Optional processed event

update_sequential_conditions(*state: sismic.model.elements.StateMixin*) → typing.Iterable[str]
Update sequential conditions, and return a list of already unsatisfied conditions.

Parameters **state** – for given state

Returns a list of already unsatisfied conditions.

class sismic.code.**PythonEvaluator**(*interpreter=None*, *, *initial_context: typing.Mapping[str, typing.Any] = None*) → None
Bases: sismic.code.evaluator.Evaluator

A code evaluator that understands Python.

Depending on the method that is called, the context can expose additional values:

•On both code execution and code evaluation:

- A *time: float* value that represents the current time exposed by the interpreter.
- An *active(name: str) -> bool* Boolean function that takes a state name and return *True* if and only if this state is currently active, ie. it is in the active configuration of the Interpreter instance that makes use of this evaluator.

•On code execution:

- A *send(name: str, **kwargs) -> None* function that takes an event name and additional keyword parameters and raises an internal event with it.
- If the code is related to a transition, the *event: Event* that fires the transition is exposed.

•On guard or contract evaluation:

- If the code is related to a transition, the *event: Event* that fires the transition is exposed.

•On guard or contract (except preconditions) evaluation:

- An *after(sec: float) -> bool* Boolean function that returns *True* if and only if the source state was entered more than *sec* seconds ago. The time is evaluated according to Interpreter's clock.
- A *idle(sec: float) -> bool* Boolean function that returns *True* if and only if the source state did not fire a transition for more than *sec* ago. The time is evaluated according to Interpreter's clock.

•On contract (except preconditions) evaluation:

- A variable *__old__* that has an attribute *x* for every *x* in the context when either the state was entered (if the condition involves a state) or the transition was processed (if the condition involves a transition). The value of *__old__.x* is a shallow copy of *x* at that time.

•On contract evaluation:

- A *sent(name: str) -> bool* function that takes an event name and return True if an event with the same name was sent during the current step.
- A *received(name: str) -> bool* function that takes an event name and return True if an event with the same name is currently processed in this step.

If an exception occurred while executing or evaluating a piece of code, it is propagated by the evaluator.

Each piece of code is executed with (a partially isolated) local context. Every state and every transition has a specific execution context. The code associated with a state is executed in a local context which is composed of local variables and every variable that is defined in the context of the parent state (and so one until the root context is reached). The context of a transition is built upon the context of its source state. The specific context of a state is available through the *context_for* method of a PythonEvaluator.

Parameters

- **interpreter** – the interpreter that will use this evaluator, is expected to be an *Interpreter* instance
- **initial_context** – a dictionary that will be used as *__locals__*

context_for(*name: str*) → sismic.code.python.Context
Context object for given state name.

Parameters **name** – State name

Returns Context object

evaluate_guard(*transition: sismic.model.elements.Transition*, *event: sismic.model.events.Event*) → bool
Evaluate the guard for given transition.

Parameters

- **transition** – the considered transition
- **event** – instance of *Event* if any

Returns truth value of *code*

evaluate_invariants(*obj*, *event: sismic.model.events.Event = None*) → typing.Iterator[str]
Evaluate the invariants for given object (either a *StateMixin* or a *Transition*) and return a list of conditions that are not satisfied.

Parameters

- **obj** – the considered state or transition
- **event** – an optional *Event* instance, in the case of a transition

Returns list of unsatisfied conditions

evaluate_postconditions(*obj*, *event: sismic.model.events.Event = None*) → typing.Iterator[str]
Evaluate the postconditions for given object (either a *StateMixin* or a *Transition*) and return a list of conditions that are not satisfied.

Parameters

- **obj** – the considered state or transition
- **event** – an optional *Event* instance, in the case of a transition

Returns list of unsatisfied conditions

evaluate_preconditions(*obj*, *event: sismic.model.events.Event = None*) → typing.Iterator[str]
Evaluate the preconditions for given object (either a *StateMixin* or a *Transition*) and return a list of conditions that are not satisfied.

Parameters

- **obj** – the considered state or transition
- **event** – an optional *Event* instance, in the case of a transition

Returns list of unsatisfied conditions

evaluate_sequential_conditions(*state: sismic.model.elements.StateMixin*) → typing.Iterable[str]

Evaluate sequential conditions, and return a list of unsatisfied conditions.

Parameters **state** – for given state

Returns a list of unsatisfied conditions.

execute_action(*transition: sismic.model.elements.Transition*, *event: sismic.model.events.Event*) → typing.List[sismic.model.events.Event]

Execute the action for given transition. This method is called for every transition that is processed, even those with no *action*.

Parameters

- **transition** – the considered transition
- **event** – instance of *Event* if any

Returns a list of sent events

execute_onentry(*state: sismic.model.elements.StateMixin*) → typing.List[sismic.model.events.Event]

Execute the on entry action for given state. This method is called for every state that is entered, even those with no *on_entry*.

Parameters **state** – the considered state

Returns a list of sent events

execute_onexit(*state: sismic.model.elements.StateMixin*) → typing.List[sismic.model.events.Event]

Execute the on exit action for given state. This method is called for every state that is exited, even those with no *on_exit*.

Parameters **state** – the considered state

Returns a list of sent events

execute_statechart(*statechart: sismic.model.statechart.Statechart*) → typing.List[sismic.model.events.Event]

Execute the initial code of a statechart. This method is called at the very beginning of the execution.

Parameters **statechart** – statechart to consider

Returns a list of sent events

initialize_sequential_conditions(*state: sismic.model.elements.StateMixin*) → None

Initialize sequential conditions.

Parameters **state** – for given state.

on_step_starts(*event: sismic.model.events.Event = None*) → None

Called each time the interpreter starts a macro step.

Parameters **event** – Optional processed event

update_sequential_conditions(*state: sismic.model.elements.StateMixin*) → typing.Iterable[str]

Update sequential conditions, and return a list of already unsatisfied conditions.

Parameters **state** – for given state

Returns a list of already unsatisfied conditions.

Module *exceptions*

exception sismic.exceptions.**StatechartError**
Bases: sismic.exceptions.SismicError

Base error for anything that is related to a statechart.

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception sismic.exceptions.**CodeEvaluationError**
Bases: sismic.exceptions.SismicError

Base error for anything related to the evaluation of the code contained in a statechart.

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception sismic.exceptions.**ExecutionError**
Bases: sismic.exceptions.SismicError

Base error for anything related to the execution of a statechart.

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception sismic.exceptions.**ConflictingTransitionsError**
Bases: *sismic.exceptions.ExecutionError*

When multiple conflicting (parallel) transitions can be processed at the same time.

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception sismic.exceptions.**NonDeterminismError**
Bases: *sismic.exceptions.ExecutionError*

In case of non-determinism.

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception sismic.exceptions.**ContractError**(*configuration=None*, *step=None*, *obj=None*, *assertion=None*, *context=None*)
Bases: sismic.exceptions.SismicError

Base exception for situations in which a contract is not satisfied. All the parameters are optional, and are exposed to ease debug.

Parameters

- **configuration** – list of active states
- **step** – a *MicroStep* or *MacroStep* instance.
- **obj** – the object that is concerned by the assertion
- **assertion** – the assertion that failed
- **context** – the context in which the condition failed

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception sismic.exceptions.**PreconditionError**(*configuration=None*, *step=None*, *obj=None*, *assertion=None*, *context=None*)

Bases: *sismic.exceptions.ContractError*

A precondition is not satisfied.

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception sismic.exceptions.**PostconditionError**(*configuration=None*, *step=None*, *obj=None*, *assertion=None*, *context=None*)

Bases: *sismic.exceptions.ContractError*

A postcondition is not satisfied.

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception sismic.exceptions.**InvariantError**(*configuration=None*, *step=None*, *obj=None*, *assertion=None*, *context=None*)

Bases: *sismic.exceptions.ContractError*

An invariant is not satisfied.

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception sismic.exceptions.**SequentialConditionError**(*configuration=None*, *step=None*, *obj=None*, *assertion=None*, *context=None*)

Bases: *sismic.exceptions.ContractError*

A sequential condition is not satisfied.

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

Module *interpreter*

class sismic.interpreter.**Interpreter**(*statechart: sismic.model.statechart.Statechart, *, evaluator_klass: typing.Callable[[typing.Interpreter], sismic.code.evaluator.Evaluator] = <class 'sismic.code.python.PythonEvaluator'>, initial_context: typing.Mapping[str, typing.Any] = None, ignore_contract: bool = False*) → None

Bases: object

A discrete interpreter that executes a statechart according to a semantic close to SCXML.

Parameters

- **statechart** – statechart to interpret
- **evaluator_klass** – An optional callable (eg. a class) that takes an interpreter and an optional initial context as input and return an *Evaluator* instance that will be used to initialize the interpreter. By default, the *PythonEvaluator* class will be used.
- **initial_context** – an optional initial context that will be provided to the evaluator. By default, an empty context is provided
- **ignore_contract** – set to True to ignore contract checking during the execution.

bind(*interpreter_or_callable: typing.Union[typing.Interpreter, typing.Callable[[sismic.model.events.Event], typing.Any]]*) → sismic.interpreter.interpreter.Interpreter

Bind an interpreter or a callable to the current interpreter. Each time an internal event is sent by this interpreter, any bound object will be called with the same event. If *interpreter_or_callable* is an *Interpreter* instance, its *queue* method is called. This is, if *i1* and *i2* are interpreters, *i1.bind(i2)* is equivalent to *i1.bind(i2.queue)*.

Parameters **interpreter_or_callable** – interpreter or callable to bind

Returns *self* so it can be chained

configuration

List of active states names, ordered by depth. Ties are broken according to the lexicographic order on the state name.

context

The context of execution.

execute(*max_steps: int = -1*) → typing.List[sismic.model.steps.MacroStep]

Repeatedly calls *execute_once* and return a list containing the returned values of *execute_once*.

Notice that this does NOT return an iterator but computes the whole list first before returning it.

Parameters **max_steps** – An upper bound on the number steps that are computed and returned. Default is -1, no limit. Set to a positive integer to avoid infinite loops in the statechart execution.

Returns A list of *MacroStep* instances

execute_once() → typing.Union[sismic.model.steps.MacroStep, NoneType]

Processes a transition based on the oldest queued event (or no event if an eventless transition can be processed), and stabilizes the interpreter in a stable situation (ie. processes initial states, history states, etc.). When multiple transitions are selected, they are atomically processed: states are exited, transition is processed, states are entered, statechart is stabilized and only after that, the next transition is processed.

Returns a macro step or *None* if nothing happened

final

Boolean indicating whether this interpreter is in a final configuration.

queue(*event: sismic.model.events.Event*) → sismic.interpreter.interpreter.Interpreter

Queue an event to the interpreter.

Parameters **event** – an *Event* instance.

Returns *self* so it can be chained.

raise_event(*event: sismic.model.events.Event*) → None

Raise an event from the statechart. Events are propagated to bound interpreters as non-internal events, and added to the internal queue of the current interpreter.

Parameters **event** – raised event.

statechart

Embedded statechart

time

Time value (in seconds) for the internal clock

Module *io*

sismic.io.**import_from_yaml**(*statechart: typing.Iterable[str], ignore_schema: bool = False, ignore_validation: bool = False*) → sismic.model.statechart.Statechart

Import a statechart from a YAML representation.

Unless specified, the structure contained in the YAML is validated against a predefined schema (see *sismic.io.SCHEMA*), and the resulting statechart is validated using its *validate()* method.

Parameters

- **statechart** – string or any equivalent object
- **ignore_schema** – set to *True* to disable yaml validation.
- **ignore_validation** – set to *True* to disable statechart validation.

Returns a *Statechart* instance

sismic.io.**export_to_yaml**(*statechart: sismic.model.statechart.Statechart*) → str

Export given *Statechart* instance to YAML

Parameters **statechart** –

Returns A textual YAML representation

Module *model*

class sismic.model.**ActionStateMixin**(*on_entry: str = None, on_exit: str = None*) → None

Bases: object

State that can define actions on entry and on exit.

Parameters

- **on_entry** – code to execute when state is entered
- **on_exit** – code to execute when state is exited

class sismic.model.**BasicState**(*name: str, on_entry: str = None, on_exit: str = None*) → None

Bases: sismic.model.elements.ContractMixin, sismic.model.elements.StateMixin, sismic.model.elements.ActionStateMixin, sismic.model.elements.TransitionStateMixin

A basic state, with a name, transitions, actions, etc. but no child state.

Parameters

- **name** – name of this state
- **on_entry** – code to execute when state is entered
- **on_exit** – code to execute when state is exited

class sismic.model.**CompositeStateMixin**

Bases: object

Composite state can have children states.

class sismic.model.**CompoundState**(*name: str, initial: str = None, on_entry: str = None, on_exit: str = None*) → None

Bases: sismic.model.elements.ContractMixin, sismic.model.elements.StateMixin, sismic.model.elements.ActionStateMixin, sismic.model.elements.TransitionStateMixin, sismic.model.elements.CompositeStateMixin

Compound states must have children states.

Parameters

- **name** – name of this state
- **initial** – name of the initial state
- **on_entry** – code to execute when state is entered
- **on_exit** – code to execute when state is exited

class sismic.model.**ContractMixin** → None
Bases: object

Mixin with a contract: preconditions, postconditions, invariants and sequences.

class sismic.model.**DeepHistoryState**(*name: str*, *on_entry: str = None*, *on_exit: str = None*, *memory: str = None*) → None
Bases: sismic.model.elements.ContractMixin, sismic.model.elements.StateMixin, sismic.model.elements.ActionStateMixin, sismic.model.elements.HistoryStateMixin

A deep history state resumes the execution of its parent, and of every nested active states in its parent.

Parameters

- **name** – name of this state
- **on_entry** – code to execute when state is entered
- **on_exit** – code to execute when state is exited
- **memory** – name of the initial state

class sismic.model.**Event**(*name: str*, ***additional_parameters: typing.Any*) → None
Bases: object

Simple event with a name and (optionally) some data. Unless the attribute already exists, each key from *data* is exposed as an attribute of this class.

The list of defined attributes can be obtained using *dir(event)*.

Parameters

- **name** – Name of the event
- **data** – additional data (mapping, dict-like)

class sismic.model.**FinalState**(*name: str*, *on_entry: str = None*, *on_exit: str = None*) → None
Bases: sismic.model.elements.ContractMixin, sismic.model.elements.StateMixin, sismic.model.elements.ActionStateMixin

Final state has NO transition and is used to detect state machine termination.

Parameters

- **name** – name of this state
- **on_entry** – code to execute when state is entered
- **on_exit** – code to execute when state is exited

class sismic.model.**HistoryStateMixin**(*memory: str = None*) → None
Bases: object

History state has a memory that can be resumed.

Parameters **memory** – name of the initial state

class sismic.model.**InternalEvent**(*name: str*, ***additional_parameters: typing.Any*) → None
Bases: sismic.model.events.Event

Subclass of Event that represents an internal event.

class sismic.model.**MacroStep**(*time: float, steps: typing.List[sismic.model.steps.MicroStep]*) → None
Bases: object

A macro step is a list of micro steps.

Parameters

- **time** – the time at which this step was executed
- **steps** – a list of *MicroStep* instances

entered_states
List of the states names that were entered.

event
Event (or *None*) that was consumed.

exited_states
List of the states names that were exited.

sent_events
List of events that were sent during this step.

steps
List of micro steps

time
Time at which this step was executed.

transitions
A (possibly empty) list of transitions that were triggered.

class sismic.model.**MicroStep**(*event: sismic.model.events.Event = None*, *transition: sismic.model.elements.Transition = None*, *entered_states: typing.List[str] = None*, *exited_states: typing.List[str] = None*, *sent_events: typing.List[sismic.model.events.Event] = None*) → None
Bases: object

Create a micro step.

A step consider *event*, takes a *transition* and results in a list of *entered_states* and a list of *exited_states*. Order in the two lists is REALLY important!

Parameters

- **event** – Event or None in case of eventless transition
- **transition** – a *Transition* or None if no processed transition
- **entered_states** – possibly empty list of entered states
- **exited_states** – possibly empty list of exited states
- **sent_events** – a possibly empty list of events that are sent during the step

class sismic.model.**OrthogonalState**(*name: str*, *on_entry: str = None*, *on_exit: str = None*) → None
Bases: sismic.model.elements.ContractMixin, sismic.model.elements.StateMixin, sismic.model.elements.ActionStateMixin, sismic.model.elements.TransitionStateMixin, sismic.model.elements.CompositeStateMixin

Orthogonal states run their children simultaneously.

Parameters

- **name** – name of this state
- **on_entry** – code to execute when state is entered
- **on_exit** – code to execute when state is exited

class sismic.model.**ShallowHistoryState**(*name: str*, *on_entry: str = None*, *on_exit: str = None*, *memory: str = None*) → None
Bases: sismic.model.elements.ContractMixin, sismic.model.elements.StateMixin, sismic.model.elements.ActionStateMixin, sismic.model.elements.HistoryStateMixin

A shallow history state resumes the execution of its parent. It activates the latest visited state of its parent.

Parameters

- **name** – name of this state
- **on_entry** – code to execute when state is entered
- **on_exit** – code to execute when state is exited
- **memory** – name of the initial state

class sismic.model.**StateMixin**(*name: str*) → None
Bases: object

State element with a name.

Parameters **name** – name of the state

class sismic.model.**Statechart**(*name: str*, *description: str = None*, *preamble: str = None*) → None
Bases: object

Python structure for a statechart

Parameters

- **name** – Name of this statechart
- **description** – optional description
- **preamble** – code to execute to bootstrap the statechart

add_state(*state: sismic.model.elements.StateMixin, parent: typing.Union[str, NoneType]*) → None
Add given state (a *StateMixin* instance) on given parent (its name as an *str*). If given state should be use as a root state, set *parent* to None.

Parameters

- **state** – state to add
- **parent** – name of its parent, or None

Raises ***StatechartError*** –

add_transition(*transition: sismic.model.elements.Transition*) → None
Register given transition and register it on the source state

Parameters **transition** – transition to add

Raises ***StatechartError*** –

ancestors_for(*name: str*) → typing.List[str]
Return an ordered list of ancestors for the given state. Ancestors are ordered by decreasing depth.

Parameters **name** – name of the state

Returns state's ancestors

Raises ***StatechartError*** – if state does not exist

children_for(*name: str*) → typing.List[str]
Return the names of the children of the given state.

Parameters **name** – a state name

Returns a (possibly empty) list of children

Raises ***StatechartError*** – if state does not exist

copy_from_statechart(*statechart: sismic.model.statechart.Statechart, source: str, *, replace: str, renaming_func: typing.Callable[[str], str] = <function Statechart.<lambda>>*) → None
Copy (a part of) given *statechart* into current one.

Copy *source* state, all its descendants and all involved transitions from *statechart* into current statechart. The *source* state will override *replace* state (but will be renamed to *replace*), and all its descendants in *statechart* will be copied into current statechart. All the transitions that are involved in the process must be fully contained in *source* state (ie. for all transition T: S->T, if S (resp. T) is a descendant-or-self of *source*, then T (resp. S) must be a descendant-or-self of *source*).

If necessary, callable *renaming_func* can be provided. This function should accept a (state) name and return a (new state) name. Use *renaming_func* to avoid conflicting names in target statechart.

Parameters

- **statechart** – Source statechart from which states will be copied.
- **source** – Name of the source state.
- **replace** – Name of the target state. Should refer to a StateMixin with no child.
- **renaming_func** – Optional callable to resolve conflicting names.

depth_for(*name: str*) → int
Return the depth of given state (1-indexed).

Parameters **name** – name of the state

Returns state depth

Raises ***StatechartError*** – if state does not exist

descendants_for(*name: str*) → typing.List[str]
Return an ordered list of descendants for the given state. Descendants are ordered by increasing depth.

Parameters **name** – name of the state

Returns state's descendants

Raises ***StatechartError*** – if state does not exist

events_for(*name_or_names: typing.Union[str, typing.List[str]] = None*) → typing.List[str]
Return a list containing the name of every event that guards a transition in this statechart.

If *name_or_names* is specified, it must be the name of a state (or a list of such names). Only transitions that have a source state from this list will be considered. By default, the list contains all the states.

Parameters **name_or_names** – *None*, a state name or a list of state names.

Returns A list of event names

leaf_for (*names: typing.Iterable[str]*) → typing.List[str]
Return the leaves of *names*.

Considering the list of states names in *names*, return a list containing each element of *names* such that this element has no descendant in *names*.

Parameters **names** – a list of state names

Returns the names of the leaves in *names*

Raises ***StatechartError*** – if a state does not exist

least_common_ancestor (*name_first: str*, *name_second: str*) → str
Return the deepest common ancestor for *s1* and *s2*, or *None* if there is no common ancestor except root (top-level) state.

Parameters

- **name_first** – name of first state
- **name_second** – name of second state

Returns name of deepest common ancestor or *None*

Raises ***StatechartError*** – if state does not exist

move_state (*name: str*, *new_parent: str*) → None
Move given state (and its children) such that its new parent is *new_parent*.

Notice that a state cannot be moved inside itself or inside one of its descendants. If the state to move is the target of an *initial* or *memory* property of its parent, this property will be set to None. The same occurs if given state is an history state.

Parameters

- **name** – name of the state to move
- **new_parent** – name of the new parent

parent_for (*name: str*) → str
Return the name of the parent of given state name.

Parameters **name** – a state name

Returns its parent name, or None.

Raises ***StatechartError*** – if state does not exist

preamble
Preamble code

remove_state (*name: str*) → None
Remove given state.

The transitions that involve this state will also be removed. If the state is the target of an *initial* or *memory* property, their value will be set to None. If the state has children, they will be removed too.

Parameters **name** – name of a state

Raises ***StatechartError*** –

remove_transition (*transition: sismic.model.elements.Transition*) → None
Remove given transitions.

Parameters **transition** – a *Transition* instance

Raises ***StatechartError*** – if transition is not registered

rename_state(*old_name: str*, *new_name: str*) → None
Change state name, and adapt transitions, initial state, memory, etc.

Parameters

- **old_name** – old name of the state
- **new_name** – new name of the state

root
Root state name

rotate_transition(*transition: sismic.model.elements.Transition*, *new_source: str = ''*, *new_target: typing.Union[str, NoneType] = ''*) → None
Rotate given transition.

You MUST specify either *new_source* (a valid state name) or *new_target* (a valid state name or None) or both.

Parameters

- **transition** – a *Transition* instance
- **new_source** – a state name
- **new_target** – a state name or None

Raises ***StatechartError*** – if given transition or a given state does not exist.

state_for(*name: str*) → sismic.model.elements.StateMixin
Return the state instance that has given name.

Parameters **name** – a state name

Returns a *StateMixin* that has the same name or None

Raises ***StatechartError*** – if state does not exist

states
List of state names in lexicographic order.

transitions
List of available transitions

transitions_from(*source: str*) → typing.List[sismic.model.elements.Transition]
Return the list of transitions whose source is given name.

Parameters **source** – name of source state

Returns a list of *Transition* instances

Raises ***StatechartError*** – if state does not exist

transitions_to(*target: str*) → typing.List[sismic.model.elements.Transition]
Return the list of transitions whose target is given name. Internal transitions are returned too.

Parameters **target** – name of target state

Returns a list of *Transition* instances

Raises ***StatechartError*** – if state does not exist

transitions_with(*event: str*) → typing.List[sismic.model.elements.Transition]
Return the list of transitions that can be triggered by given event name.

Parameters **event** – name of the event

Returns a list of *Transition* instances

validate() → bool
Checks that every *CompoundState*'s initial state refer to one of its children Checks that every *HistoryStateMixin*'s memory refer to one of its parent's children

Returns True

Raises ***StatechartError*** –

class sismic.model.**Transition**(*source: str*, *target: str = None*, *event: str = None*, *guard: str = None*, *action: str = None*) → None
Bases: sismic.model.elements.ContractMixin

Represent a transition from a source state to a target state.

A transition can be eventless (no event) or internal (no target). A condition (code as string) can be specified as a guard.

Parameters

- **source** – name of the source state
- **target** – name of the target state (if transition is not internal)
- **event** – event name (if any)
- **guard** – condition as code (if any)
- **action** – action as code (if any)

eventless
Boolean indicating whether this transition is an eventless transition.

internal
Boolean indicating whether this transition is an internal transition.

class sismic.model.**TransitionStateMixin**
Bases: object

A simple state can host transitions

Module *stories*

class sismic.stories.**Pause**(*duration: float*) → None
A convenience class to represent pause, ie. delay between sent events.

Parameters **duration** – the duration of this pause

duration
The duration of this pause

class sismic.stories.**Story**
A story is a sequence of *Event* and *Pause*.

tell(*interpreter: sismic.interpreter.interpreter.Interpreter*, **args*, ***kwargs*) → typing.List[sismic.model.steps.MacroStep]
Tells the whole story to the interpreter.

Parameters

- **interpreter** – an interpreter instance
- **args** – additional positional arguments that are passed to *interpreter.execute*.
- **kwargs** – additional keywords arguments that are passed to *interpreter.execute*.

Returns the resulting trace of execution (a list of *MacroStep*)

tell_by_step(*interpreter*, **args*, ***kwargs*) → typing.Generator[[typing.Tuple[typing.Union[sismic.model.events.Event, typing.Pause], typing.List[sismic.model.steps.MacroStep]], NoneType], NoneType]

Tells the story to the interpreter, step by step. This method returns a generator which yields the event or the pause that was told to the interpreter and the result of *interpreter.execute*.

Parameters

- **interpreter** – an interpreter instance
- **args** – additional positional arguments that are passed to *interpreter.execute*.
- **kwargs** – additional keywords arguments that are passed to *interpreter.execute*.

Returns a generator that yields (told event or pause, result of *interpreter.execute*).

append(*object*) → None – append object to end

clear() → None – remove all items from L

copy() → list – a shallow copy of L

count(*value*) → integer – return number of occurrences of value

extend(*iterable*) → None – extend list by appending elements from the iterable

index(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

insert()
L.insert(index, object) – insert object before index

pop([*index*]) → item – remove and return item at index (default last).
Raises IndexError if list is empty or index is out of range.

remove(*value*) → None – remove first occurrence of value.
Raises ValueError if the value is not present.

reverse()
L.reverse() – reverse *IN PLACE*

sort(*key=None*, *reverse=False*) → None – stable sort *IN PLACE*

sismic.stories.**random_stories_generator**(*items: typing.Sequence[typing.Union[sismic.model.events.Event, sismic.stories.Pause]]*, *length: int = None*, *number: int = None*) → typing.Generator[[sismic.stories.Story, NoneType], NoneType]

A generator that returns random stories whose elements come from *items*. Parameter *items* can be any iterable containing events and/or pauses.

Parameters

- **items** – Items to pick from
- **length** – Length of the story, or *len(items)*
- **number** – number of stories to generate (None = infinite)

Returns An infinite Story generator

sismic.stories.**story_from_trace**(*trace: typing.Iterable[sismic.model.steps.MacroStep]*) → sismic.stories.Story

Return a story that is built upon the given trace (a list of macro steps).

The story is composed of the same pauses and the same events than the ones that generated the given trace. The use case is when you want to reproduce the scenario of an observed behavior.

Notice that internal events are ignored.

Parameters **trace** – A list of *MacroStep* instances.

Returns A story

Module *testing*

class sismic.testing.**ExecutionWatcher**(*tested_interpreter: sismic.interpreter.interpreter.Interpreter*) → None

Bases: object

This can be used to associate a property statechart with a statechart under test. An instance of this class is built upon an *Interpreter* instance (the tested one).

It provides a method, namely *watch_with* which takes a property statechart (and a set of optional parameters that can be used to tune the interpreter that will be built upon this property statechart) and returns the resulting *Interpreter* instance for this tester.

If started (using *start*), whenever something happens during the execution of the interpreter under test, events are automatically sent to every associated statechart properties. Their internal clock are synchronized, and the context of the statechart under test is also exposed to the property statechart, ie. if *x* is a variable in the context of a statechart under test, then *context.x* is dynamically exposed to every associated property statechart.

Parameters **tested_interpreter** – Interpreter to watch

start() → None

Send a *started* event to the statechart properties, and starts watching the execution of the statechart under test.

stop() → None

Send a *stopped* event to the statechart properties, and stops watching the execution of the statechart under test.

watch_with(*property_statechart: sismic.model.statechart.Statechart*, *fails_fast: bool = False*, *interpreter_klass: typing.Callable[..., sismic.interpreter.interpreter.Interpreter] = <class 'sismic.interpreter.interpreter.Interpreter'>*, ***kwargs*) → sismic.interpreter.interpreter.Interpreter

Watch the execution of the tested interpreter with given sproperty statechart.

interpreter_klass is a callable that accepts a *Statechart* instance, an *initial_context* parameter and any additional parameters provided to this method. This callable must return an *Interpreter* instance

Parameters

- **property_statechart** – a property statechart (instance of *Statechart*)
- **fails_fast** – If True (default is False), the execution of the statechart under test will raise an AssertionError as soon as given property statechart reaches a final state.
- **interpreter_klass** – a callable that accepts a *Statechart* instance, an *initial_context* and any additional (optional) parameters provided to this method.

Returns the interpreter instance that wraps given property statechart.

sismic.testing.**teststory_from_trace**(*trace: typing.List[sismic.model.steps.MacroStep]*) → sismic.stories.Story

Return a test story based on the given *trace*, a list of macro steps. See documentation to see which are the events that are generated.

Notice that this function adds a *pause* if there is any delay between pairs of consecutive steps.

Parameters `trace` – a list of *MacroStep* instances

Returns A story

CHAPTER 3

Source code

The source code is available on GitHub: https://github.com/AlexandreDecan/sismic

Use GitHub's integrated services to contribute suggestions and feature requests for this library or to report bugs.

Python Module Index

S

Index

Symbols

A

B

C

D

E

F

H

I

L

M

N

O

P

Q

R

S

T

U

V

W