Model-Based Testing of Executable Statecharts

Tom Mens, Alexandre Decan
Software Engineering Lab
University of Mons, Belgium
Agile and defensive development

Many “agile” development techniques provide lightweight approaches to facilitate change and increase reliability of software

- Quality assessment (e.g. bad smells and refactoring)
- Defensive programming (e.g. design by contract)
- Test-driven development (e.g. unit testing and behavior-driven development)
- Dynamic verification of behavioural properties

We propose to raise these techniques to the level of executable (statechart) models
Future work (spoiler)

Facilitate evolution of behavioural design models

- Detecting model smells
- Model refactoring
  - E.g. splitting up a complex statechart into multiple statecharts
- Semantic variation
  - Detecting if statechart is compatible with alternative semantics
- Variability analysis
  - Consider product families (e.g. different microwave variants) and analyse commonalities and variabilities in their statechart models
- Design space exploration
  - Analyse pros and cons of syntactically different, but semantically similar statecharts
Agile and defensive modelling

• Advanced model testing (focus of this talk)
  • Contract-driven modeling
  • Test-driven modeling (unit testing and BDD for statecharts)
  • Dynamic verification (property statecharts)

• Future work
  • Model quality assessment (model smells)
  • Model quality improvement (model refactoring)
  • Model checking
  • Model variability analysis
  • Design space exploration
  • Model composition and scalability
  • Semantic variation
Running example

Microwave oven

- Door
  - opened()
  - closed()

- Power
  - reset()
  - inc()
  - dec()

- Timer
  - inc()
  - dec()
  - reset()
  - tick()

- Cooking
  - start()
  - stop()

- WeightSensor
  - item_placed()
  - item_removed()
  - beep(d : integer)

- Lamp
  - switch_on()
  - switch_off()

- Heating
  - set(power : integer)
  - on()
  - off()

- Display
  - clear()
  - set(i : integer)
  - set(s : string)

- Turntable
  - start()
  - stop()

- Beeper
  - beep(d : integer)
Use case name: Cook Food
Summary: User puts food in oven, and oven cooks food.
Assumptions: Oven has been configured with weight sensor and turntable.
Preconditions: Oven is closed and empty.
Postconditions: Oven has cooked the food. Oven is closed and empty.
Basic course of action:
1. User opens door.
2. User puts food in oven and closes door.
3. User sets cooking time via control panel.
4. User presses start button.
6. Remaining cooking time is displayed continuously.
7. System notifies user when cooking time has elapsed. Magnetron indicator light switches off.
8. User opens door, removes food from oven, and closes door.
9. System clears display and resets default values for cooking.
Use case name: Cook Food

Alternate courses:
1a: User presses start button while door is open. System does not start cooking.
3a: User presses start button while no food is in the oven. System does not start cooking.
3b: User presses start button while cooking time is zero. System does not start cooking.
Running example

Microwave oven

- **INPUT BUTTONS**
  - power +
  - power -
  - power 0
  - timer +
  - timer -
  - timer 0
  - start
  - stop

- **SENSORS**
  - tick
  - place item
  - remove item
  - open door
  - close door

- **COMPONENTS**
  - Display
    - TIMER: 5
  - Lamp
    - on
  - Heating
    - 700
    - on
  - Beeper
    - 3
  - Turntable
    - on
  - Controller
    - M.power: 700
    - M.timer: 5
    - controller door closed
    - closed with item cooking mode
Oven controller statechart
Software-controlled systems are **difficult to develop**

Control software can be very **complex**

- Continuous interaction between software and hardware
- Continuous interaction with external world and users
- Must respect *functional* requirements
  - Oven should cook food placed in oven with specified power and duration
- Must respect *non-functional* requirements
  - Oven should stop sending microwaves if doors are opened
Contract-driven development

• Add precise and dynamically verifiable specifications to executable software components (e.g., methods, functions, classes)

• Based on Bertrand Meyer’s “Design by Contract”

• The software component should respect a contract, composed of
  – preconditions
  – postconditions
  – invariants
Example (taken from www.eiffel.com/values/design-by-contract/introduction)

```eiffel
class DICTIONARY [ ELEMENT ]
  feature
    put (x : ELEMENT; key : STRING ) is
      require
        count <= capacity
        not key.empty
      ensure
        has (x)
        item (key) = x
        count = old count + 1
      end
  invariant
    0 <= count
    count <= capacity
  end
```

Contract-driven modelling

Contracts for microwave controller

context controller
inv: not sent(heating_on) or active(cooking mode)
inv: timer>=0
inv: 0 < power <= MAXPOWER

context cooking mode
pre: timer>0
inv: timer >= 0
inv: power == power@pre
post: received(door_opened) or timer==0

context ready
inv: timer > 0
Telling stories

Story(
  event door_opened,
  event item_placed,
  event door_closed,
  event timer_dec
).tell(interpreter)
Example of failing contract

InvariantError
State: controller
Assertion: timer >= 0
Configuration:
[controller, door closed, closed with item, program mode, not ready]
Step:
event timer_dec
internal transition on closed with item
Solution to failing contract

Add guards to the actions associated to the events that increment and decrement power and timer

\[
\text{timer} \_\text{dec\ [timer}>0] / \text{timer} -= 1
\]

\[
\text{power} \_\text{inc\ [power}<\text{MAXPOWER}] / \text{power} += 1
\]

\[
\text{power} \_\text{dec\ [power}>1] / \text{power} -= 1
\]
Test-driven development

test negative_timer:
    Story(door_opened, itemplaced, door_closed, timer_dec).tell(statechart)
    statechart.execute()
    assertsEqual(State(controller).timer, 0)

test no_heating_when_door_is_not_closed:
    Story(door_opened, item_placed, timer_inc, cooking_start).tell(statechart)
    statechart.execute()
    assertFalse active(cooking mode)
    assertFalse sent(heating_on)

Without guards on timer_dec event

test negative_timer ... FAIL
    test no_heating_when_door_is_not_closed ... ok

AssertionError: -1 != 0

Ran 2 tests in 0.005s
FAILED (failures=1)
Test-driven development

**test negative_timer:**

Story(`door_opened, item_placed, door_closed, timer_dec`).tell(statechart)
statechart.execute()
assertEqual(State(controller).timer, 0)

**test no_heating_when_door_is_not_closed:**

Story(`door_opened, item_placed, timer_inc, cooking_start`).tell(statechart)
statechart.execute()
assertFalse active(`cooking mode`)
assertFalse sent(`heating_on`)

With guards on `timer_dec` event

---
test negative_timer ... ok
test no_heating_when_door_is_not_closed ... ok
---------------------------------------------------------------------------
Ran 2 tests in 0.005s
OK
Behaviour-Driven Development

- Include customer test practices into TDD
- Encourage collaboration between developers, QA, and non-technical stakeholders (domain experts, project managers, users)
- Use a domain-specific (non-technical) language to specify how the code should behave
  - By defining feature specifications and scenarios
- Reduces the technical gap between developers and other project stakeholders
Software behaviour can be described in a domain-specific (non-technical) language suited to non-developers

- using the *Gherkin* language
- Supported by *Cucumber* framework in many languages
Behaviour-driven development

Example
(taken from docs.behat.org/en/v2.5/guides/1.gherkin.html)

**Feature:** Serve coffee
In order to earn money customers should be able to buy coffee

**Scenario:** Buy last coffee
  - **Given** there is 1 coffee left in the machine
  - **And** I have deposited 1 dollar
  - **When** I press the coffee button
  - **Then** I should be served a coffee
**Feature:** No heating if door is opened

**Scenario:** No heating when nothing is done

Given I do nothing
And I execute the statechart
Then state cooking_mode should not be active
And event heating_on should not be fired

**Scenario:** No heating when item is placed

Given I send event door_opened
When I send event item_placed
Then event heating_on should not be fired

**Scenario:** No heating when door is not closed

Given I send event door_opened
And I send event item_placed
When I send event door_closed
Then event heating_on should not be fired

---

First variant.

Still refers to specific details of the statechart (state and event names)

1 feature passed, 0 failed, 0 skipped
3 scenarios passed, 0 failed, 0 skipped
11 steps passed, 0 failed, 0 skipped, 0 undefined

Took 0m0.005s
**Feature:** No heating if door is opened

**Scenario:** No heating when nothing is done
- **When** I power up the microwave
- **Then** heating should not be on

**Scenario:** No heating when item is placed
- **Given** I open the door
- **When** I place an item
- **Then** heating should not turn on

**Scenario:** No heating when door is not closed
- **Given** I open the door
- **And** I place an item
- **When** I close the door
- **Then** heating should not turn on

---

*Second variant.*

Much closer to natural language. All statecharts-specific concepts are abstracted away.
Coverage analysis

State coverage: 81.82%
Entered states:
controller (3) | door closed (4) | door opened (2) |
closed without item (3) | opened without item (2) |
opened with item (2) | closed with item (1) |
not ready (1) | program mode (1)
Remaining states:
cooking mode | ready

Transition coverage: 16.67%
Processed transitions:
opened without item [item_placed] -> opened with item (2)
closed without item [door_opened] -> opened without item (2)
opened with item [door_closed] -> closed with item (1)
Property statecharts

Define and verify behavioural properties by

1. instrumenting the statechart interpreter
2. intercepting specific actions of statechart being executed
   - entered(<NAME OF STATE>)
   - exited(<NAME OF STATE>)
   - consumed(<NAME OF EVENT>)
   - sent(<NAME OF EVENT>)
   - ...
3. executing a *property statechart* that verifies a desirable or undesirable property
Property statecharts

<<property statechart>>
Heating must stop when door is opened

- Heating is off
  - sent(heating_off)

- Heating is on
  - sent(heating_on)
  - consumed(door_opened)

- Consumed(tick)
- Consumed(door_closed)

<<property statechart>>
Heating does not start if door is opened

- Door is closed
  - sent(heating_on)
  - consumed(door_closed)

- Door is opened
  - consumed(door_opened)

- Failure
Tool support

Sismic = Sismic Interactive Statechart Model Interpreter and Checker

– Python library available on Python Package Index (PyPI)
– released under open source licence LGPL v3
– Source code
  • github.com/AlexandreDecan/sismic
– Documentation
  • sismic.readthedocs.io
Tool support

Sismic supports all aforementioned concepts

– Statechart execution
– Design by contract
– Unit testing
– BDD
– Coverage analysis
– Property statecharts
– And more…
Sismic file format

Representing a statechart as a YAML file

```yaml
root state:
  name: controller
  contract:
    - always: not sent('heating_on') or active('cooking mode')
    - always: timer >= 0
    - always: 0 < power <= MAXPOWER
  initial: door closed
  on entry: |
    power = DEFAULT
    timer = 0
  transitions:
    - event: input_cooking_stop
      action: |
        timer = 0
```

Sismic
file format

Representing a statechart as a YAML file

```
states:
  - name: door closed
    initial: closed without item

states:
  - name: closed without item

transitions:
  - event: door_opened
    target: opened without item

- name: closed with item
  initial: program mode
  on exit: send('display_clear')

transitions:
  - event: door_opened
    target: opened with item
  - event: input_timer_inc
    action: |
      timer = timer + 1
      send('display_set', text='TIMER: %d' % timer)
```
Stepwise execution of statechart behaviour

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

with open('microwave.yaml') as f:
    statechart = import_from_yaml(f)

interpreter = Interpreter(statechart)
interpreter.execute_once()

    MacroStep(None, [], >['controller', 'door closed', 'closed without item'], <[])
interpreter.queue(Event('door_opened'))
interpreter.execute_once()

    MacroStep(Event('door_opened'), [Transition(closed without item, opened without item, door_opened)], >['door opened', 'opened without item'], <['closed without item', 'door closed'])
from sismic.stories import Story

story = Story([Event('door_opened'), Event('item_placed'), Event('door_closed'),
               Event('timer_inc'), Event('cooking_start'), Event('tick')])

trace = story.tell(interpreter)

MacroStep(None, [], >['controller', 'door closed', 'closed without item'], <[]),
MacroStep(Event(door_opened), [Transition(closed without item, opened without item, door_opened)], >['door opened', 'opened without item'], <['closed without item', 'door closed']),
MacroStep(InternalEvent(lamp_on), [], >[], <[]),
MacroStep(Event(item_placed), [Transition(opened without item, opened with item, item_placed)], >['opened with item'], <['opened without item'])),
MacroStep(Event(door_closed), [Transition(opened with item, closed with item, door_closed)], >['door closed', 'closed with item', 'program mode', 'not ready'], <['opened with item', 'door opened'])

...
Sismic
Running stories

MacroStep(InternalEvent(lamp_off), [], >[], <[]),
MacroStep(Event(timer_inc), [Transition(closed with item, None, timer_inc)], >[], <[]),
MacroStep(None, [Transition(not ready, ready, None)], >['ready'], <['not ready']),
MacroStep(InternalEvent(display_set, text=TIMER: 1), [], >[], <[]),
MacroStep(Event(cooking_start), [Transition(ready, cooking mode, cooking_start)], >['cooking mode'], <['ready', 'program mode']),
MacroStep(InternalEvent(heating_set_power, power=900), [], >[], <[]),
MacroStep(InternalEvent(heating_on), [], >[], <[]),
MacroStep(InternalEvent(lamp_on), [], >[], <[]),
MacroStep(InternalEvent(turntable_start), [], >[], <[]),
MacroStep(Event(tick), [Transition(cooking mode, None, tick)], >[], <[]),
MacroStep(None, [Transition(cooking mode, program mode, None)], >['program mode', 'not ready'], <['cooking mode']),
MacroStep(InternalEvent(display_set, text=REMAINING: 0), [], >[], <[]),
MacroStep(InternalEvent(heating_off), [], >[], <[]),
MacroStep(InternalEvent(lamp_off), [], >[], <[]), MacroStep(InternalEvent(turntable_stop), [], >[], <[]),
MacroStep(InternalEvent(beep, number=3), [], >[], <[]),
MacroStep(InternalEvent(display_set, text=DONE), [], >[], <[]])

Sismic unit testing

• Using python’s built-in `unittest` module
  
  ```
  $ python -m unittest heating_unittest.py -v
  ```

  ```python
  def test_no_heating_when_nothing_is_done(self):
      self.interpreter.execute()
      self.assertFalse(self.is_heating())

  def test_no_heating_when_item_is_placed(self):
      events = map(Event, ['door_opened', 'item_placed'])
      story = Story(events)
      story.tell(self.interpreter)
      self.interpreter.execute()
      self.assertFalse(self.is_heating())
  ```
• Using Python’s `behave` module

```
$ sismic-behave microwave.yaml --features heating.feature
```

```
from behave import given, when, then
from sismic.io import import_from_yaml
from sismic.interpreter import Interpreter
from sismic.interpreter.helpers import log_trace
from sismic.model import Event
@given('I execute the statechart')
def execute_statechart(context):
    _execute_statechart(context, force_execution=True)
@then('state {state_name} should be active')
def state_is_active(context, state_name):
    assert state_name in context._statechart.states, 'Unknown state {}'.format(state_name)
    assert state_name in context._interpreter.configuration, 'State {} is not active'.format(state_name)
```
Sismic
Regression testing

When an error is encountered (e.g. due to failing contract or bug), `story_from_trace` can reproduce the scenario of the observed behavior, which can be used as the basis of a regression test.
Sismic

Communicating statecharts

- Statecharts can *communicate* with other statecharts or external components (e.g. a user interface) by sending/receiving events
- Realised by dynamically *binding* their statechart interpreters
Example for some elevator statechart: Events sent by buttons are propagated to elevator

elevator = Interpreter(import_from_yaml(open('elevator.yaml')))
bButtons = Interpreter(import_from_yaml(open('buttons.yaml')))
bButtons.bind(elevator)
bButtons.queue(Event('floor_2_pushed'))
bButtons.execute()

  Awaiting events in buttons: [Event(button_2_pushed)]
  Awaiting events in buttons: [InternalEvent(floorSelected, floor=2)]
  Awaiting events in elevator: [Event(floorSelected, floor=2)]
elevator.execute()
print('Current floor:', elevator.context.get('current'))
  Current floor: 2
Other features

Other semantic variants of statecharts
  – outer-first instead of inner-first semantics;
  – changing priority of events
  – …

Different ways of dealing with time
  – Real time versus simulated time
Conclusion

We support various ways to test statechart models

– Using contracts
– Using unit tests
– Using domain-specific features and scenarios (BDD)
– Using property statecharts

Implemented in Sismic, an open source Python library for interpreting statecharts
Future work

• More advanced testing techniques
  – Automatic generation of contracts based on scenario specifications
  – Automatic generation of tests based on contract specifications
  – Mutation testing
  – Support for continuous integration

• Explore/compare with (dynamic?) model checking techniques
  – Based on temporal logics, labeled transition systems, …
  – Using Dwyer’s specification patterns

• And many more …
Future work

Facilitate statechart evolution

– Detecting model smells
– Model refactoring
  E.g. splitting up a complex statechart into multiple statecharts
– Semantic variation
  Detecting if statechart is compatible with alternative semantics
– Variability analysis
  Consider product families (e.g. different microwave variants) and analyse commonalities and variabilities in their statechart models
– Design space exploration
  Analyse pros and cons of syntactically different, but semantically similar statecharts