Model Testing of Executable Statecharts

Tom **Mens**

Alexandre **Decan**

Software Engineering Lab

University of Mons, Belgium

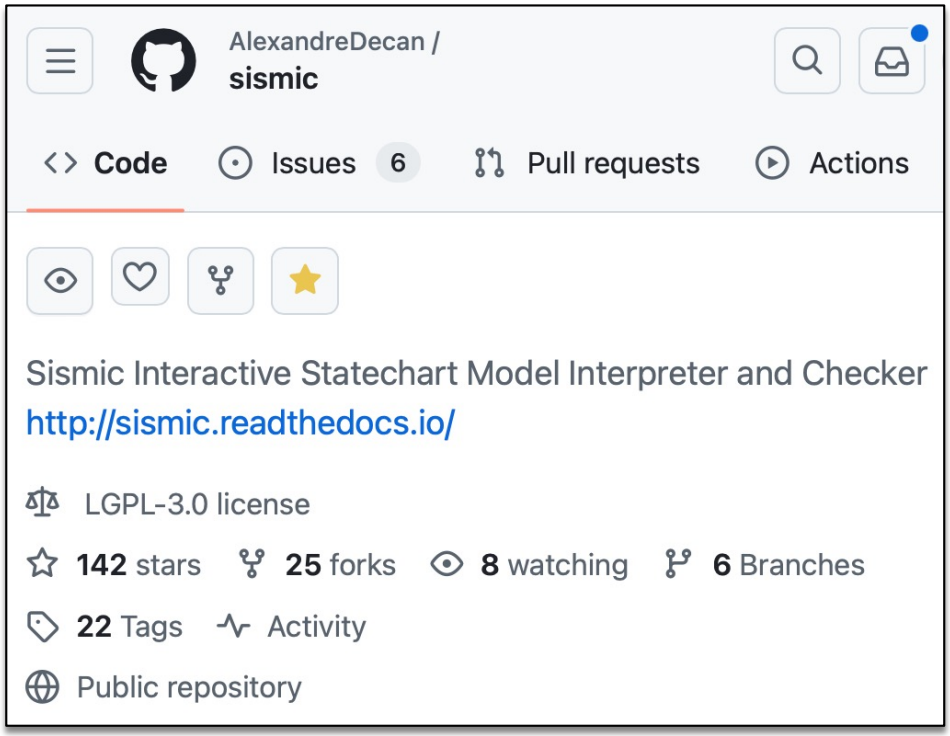
Software & Systems Modeling (2019) 18:837–863
https://doi.org/10.1007/s10270-018-0676-3

THEME SECTION PAPER

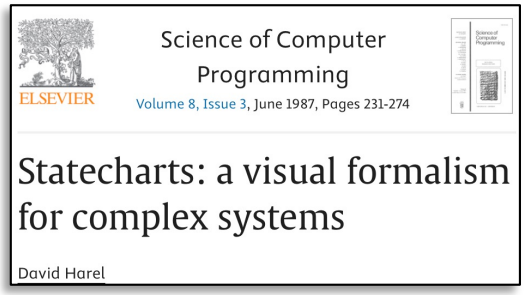
A method for testing and validating executable statechart models

Tom Mens[1] · Alexandre Decan[1] · Nikolaos I. Spanoudakis[2]

Received: 29 August 2016 / Accepted: 11 April 2018 / Published online: 3 May 2018



Statecharts (hierarchical state machines)

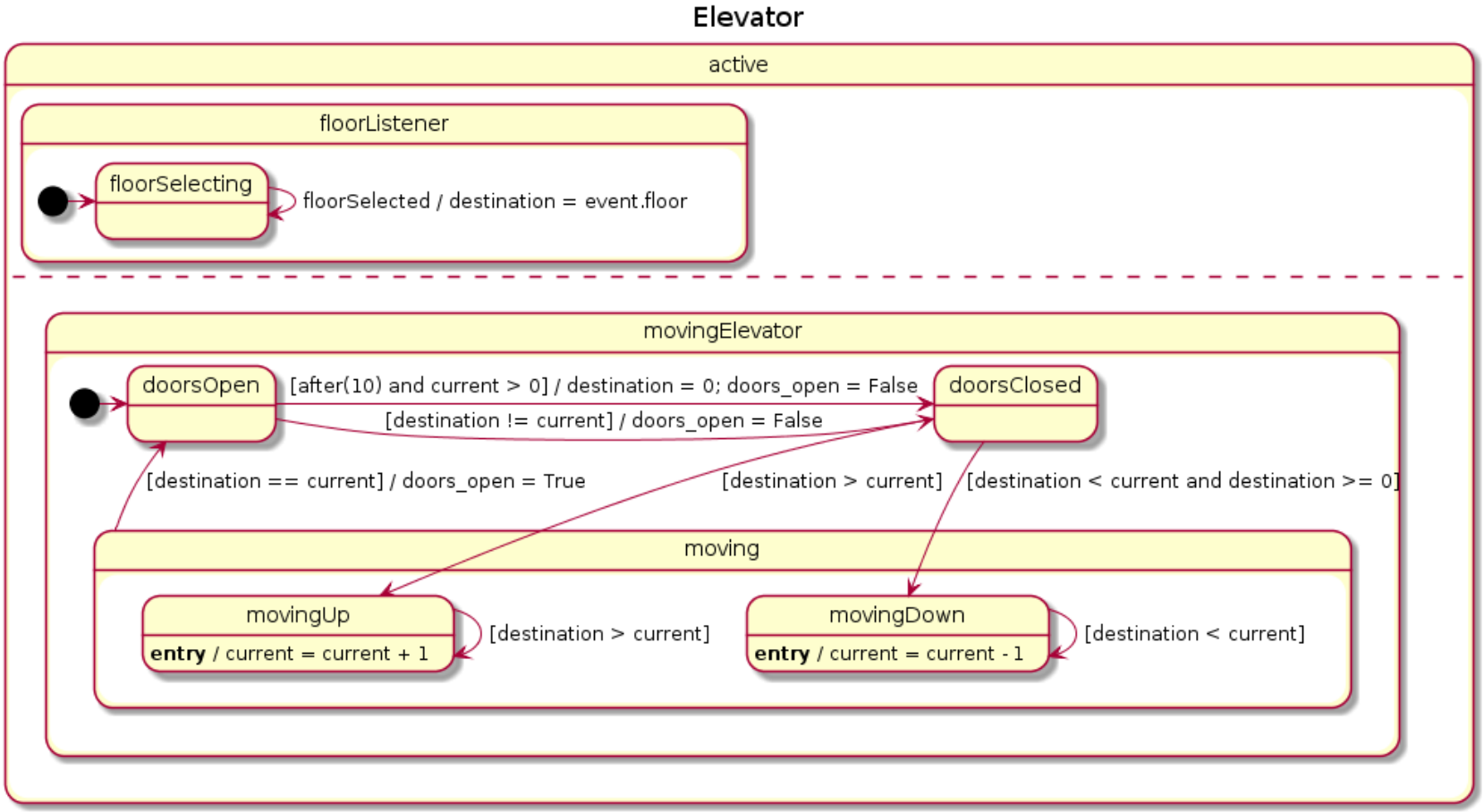


- Invented in 1987 by David Harel
- Used to specify, design and control the complex behaviour of discrete-event systems
  - embedded systems, user interfaces, network protocols, robots, computer game logic, ...
- Executable semantics
  - Can be used to generate executable source code, if a statechart compiler is available
  - Can be integrated in an existing code base directly, if a statechart interpreter is available

Statecharts
A simple example

An Elevator controller

Visualised with PlantUML (https://plantuml.com)



Statecharts
Existing Tooling

Many existing tools

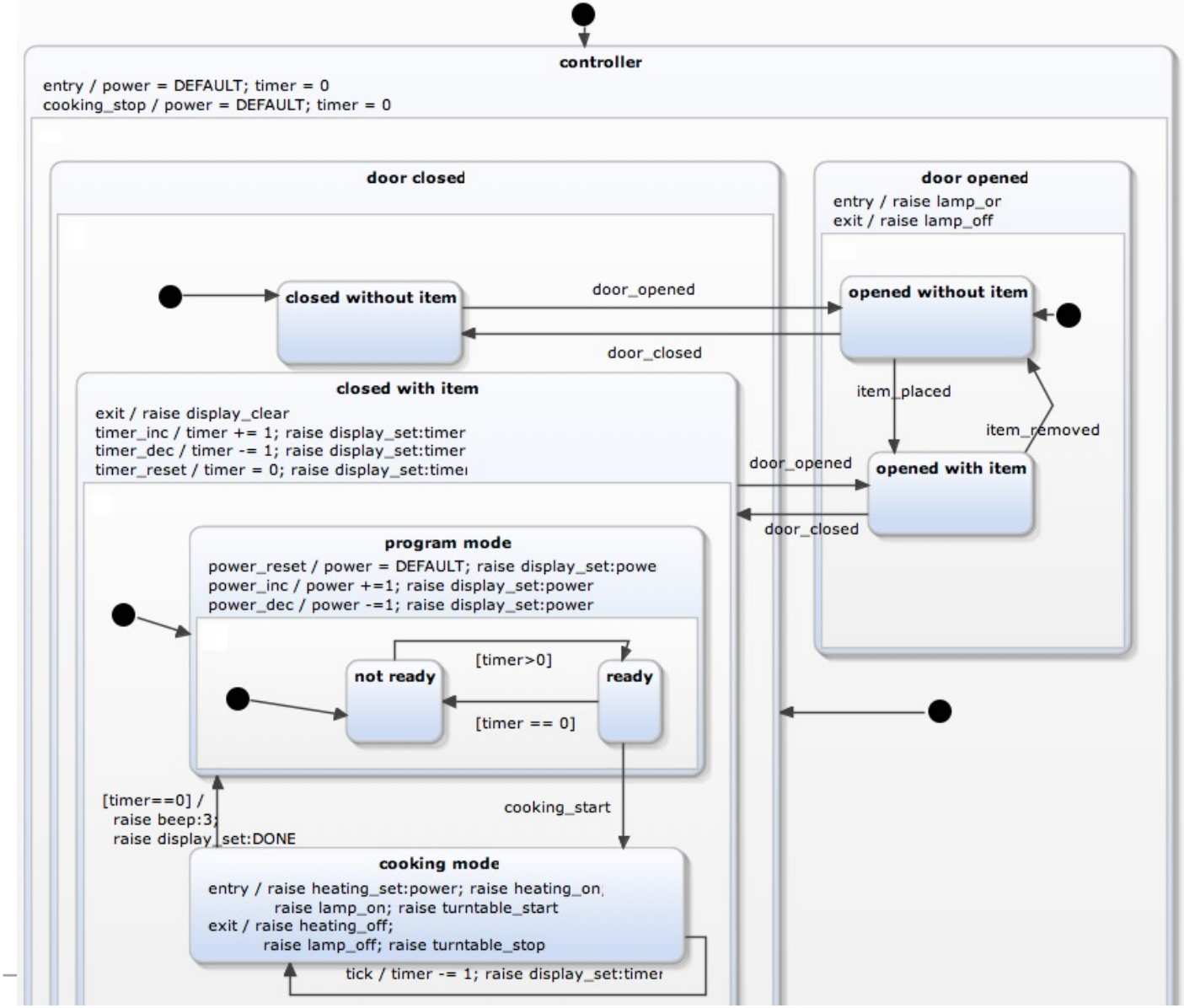
- Itemis Create: A statechart simulator and code generator
  https://www.itemis.com/en/products/itemis-create/
- Mathworks Stateflow
  https://www.mathworks.com/products/stateflow.html
- QuantumLeaps QM and QP for embedded software development
  https://www.state-machine.com
- IBM Rational Statemate
- IBM Rational Rhapsody

SISMIC: A Python-based statechart interpreter
https://pypi.org/project/sismic/

focus of this talk!

Running example
A microwave oven controller

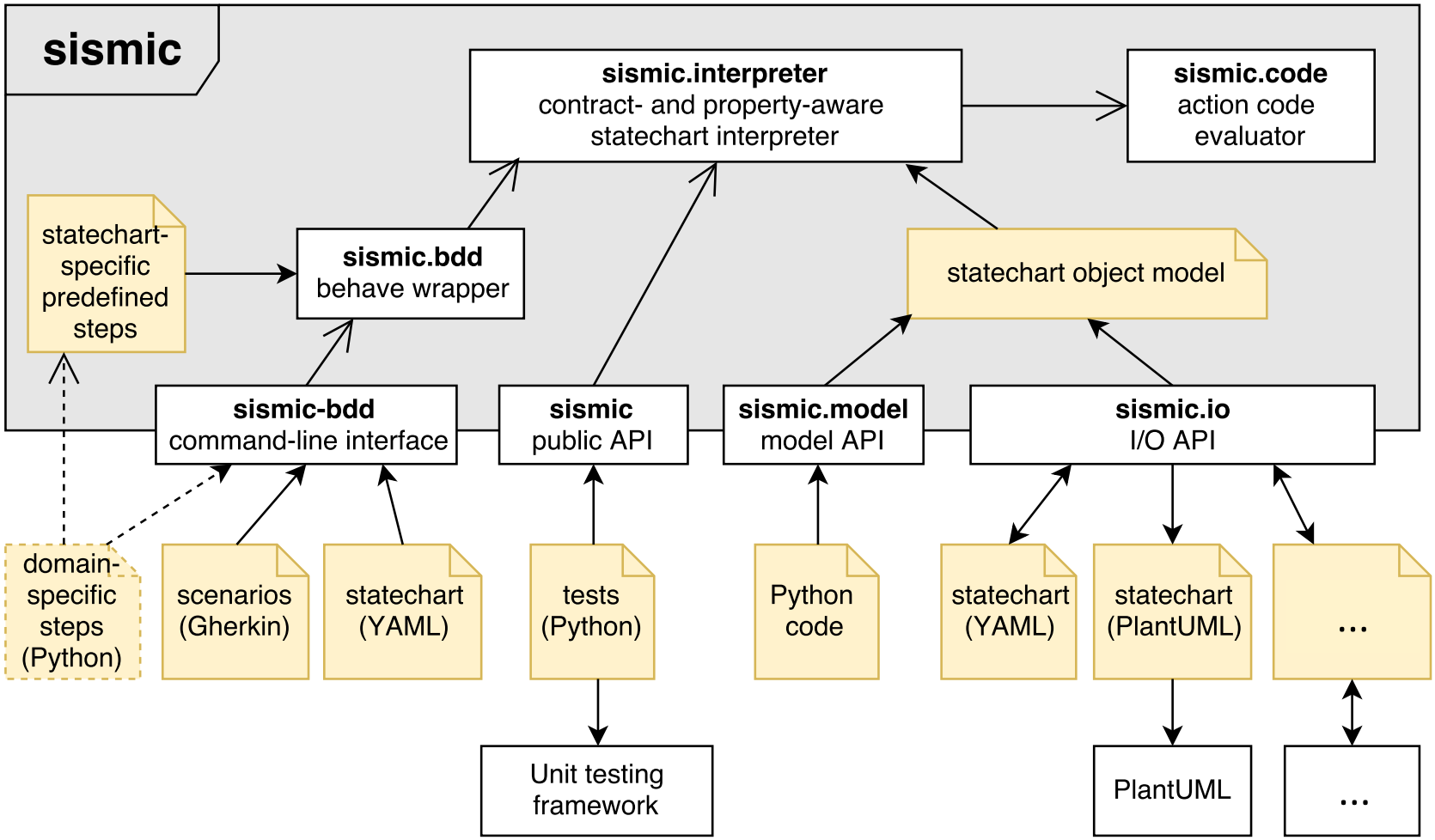
Created with
Itemis Create



Defensive software development

- Many “agile” development techniques provide lightweight approaches to facilitate change and increase reliability of software
  - Design by contract
  - Test-driven development
    - **unit testing**
    - **behavior-driven development**
  - Dynamic verification of properties
- We raise these techniques to the level of executable statechart models

SISMIC Architecture



https://sismic.readthedocs.io

https://github.com/AlexandreDecan/sismic

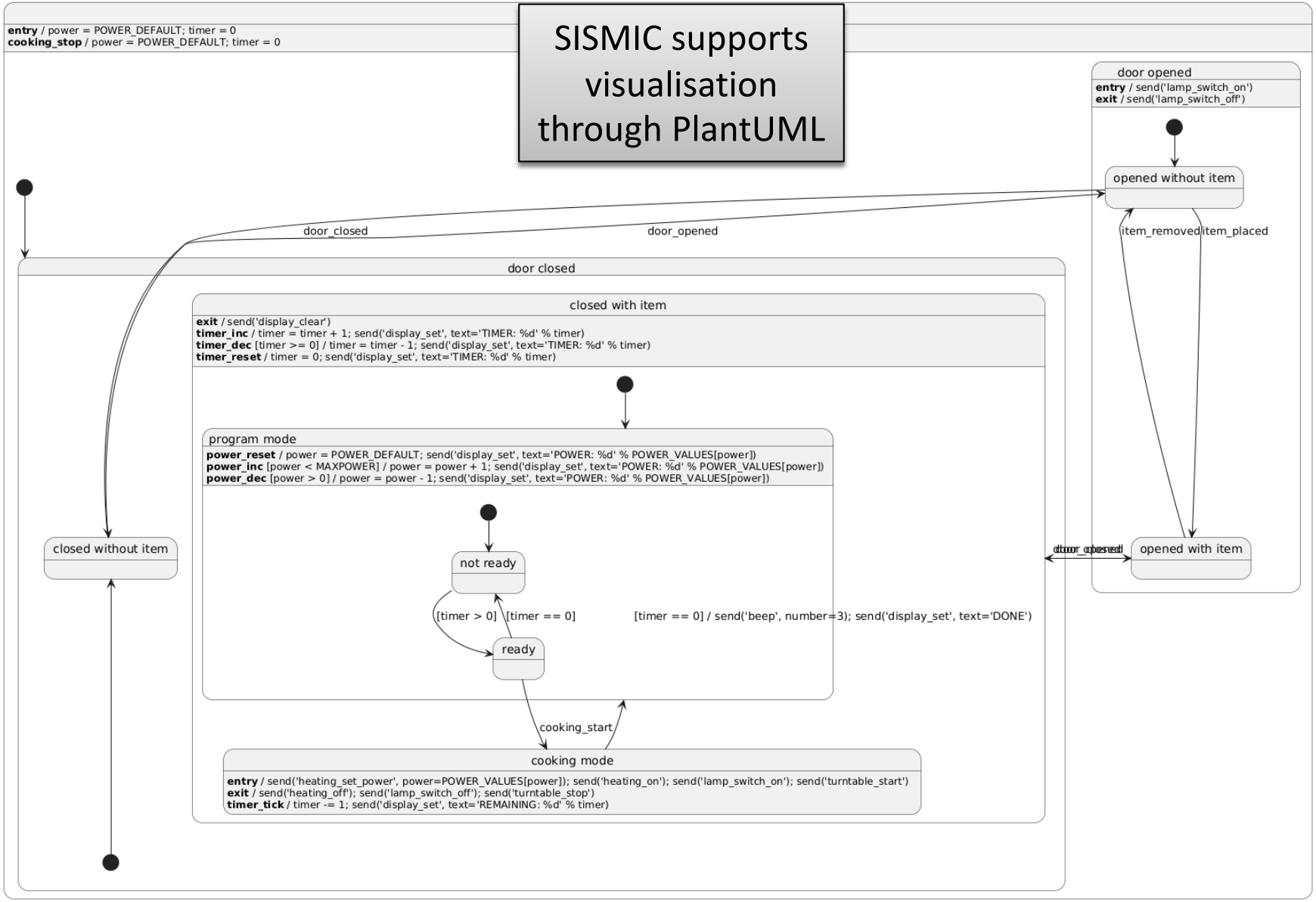
Running example
A microwave oven controller

```
statechart:
  name: Microwave controller
  preamble: |
    POWER_VALUES = [300, 600, 900, 1200, 1500]
    POWER_DEFAULT = 2 # 900W
    MAXPOWER = 3 # 1200W
  root state:
    name: controller
    initial: door closed
    on entry: |
      power = POWER_DEFAULT
      timer = 0
    transitions:
     - event: cooking_stop
       action: |
         power = POWER_DEFAULT
         timer = 0
    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
          ...
```

SISMIC adopts a YAML-based specification of statecharts

Running example
A micowave oven controller

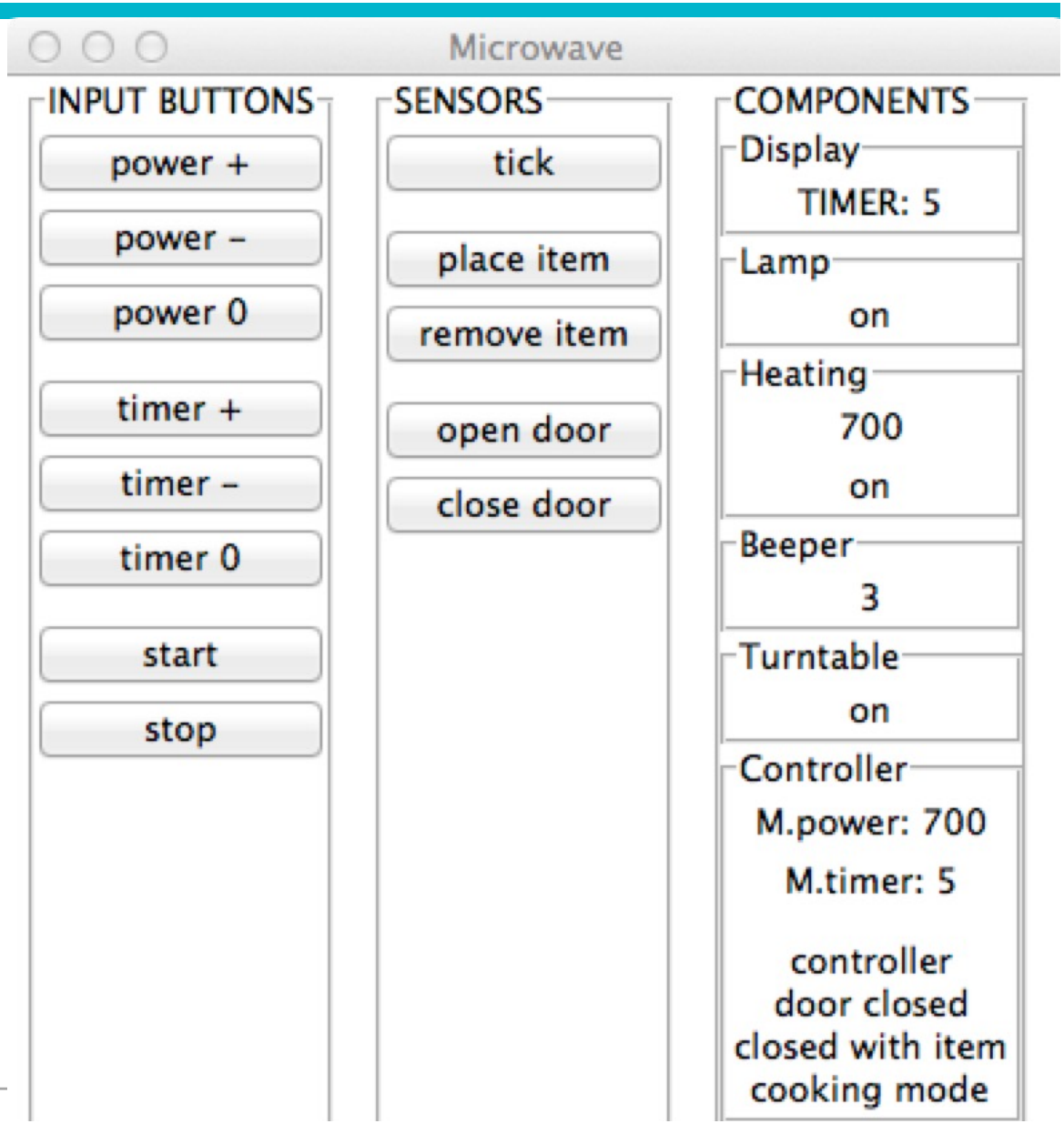
SISMIC supports visualisation through PlantUML



Running example
A microwave oven controller

SISMIC provides an API for Python code to use the statechart model

Example: Basic GUI to control the executable statechart model



SISMIC
Macrostep semantics

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'])

Software-controlled systems are *difficult to develop*

Control software can be very ***complex***

- Continuous interaction
  - between software and hardware
  - with external world and users

- Must respect
  - *functional* requirements
    - Oven should cook food placed in oven with specified power and duration
  - *non-functional* requirements
    - Oven should stop cooking if doors are opened

Contract-driven development

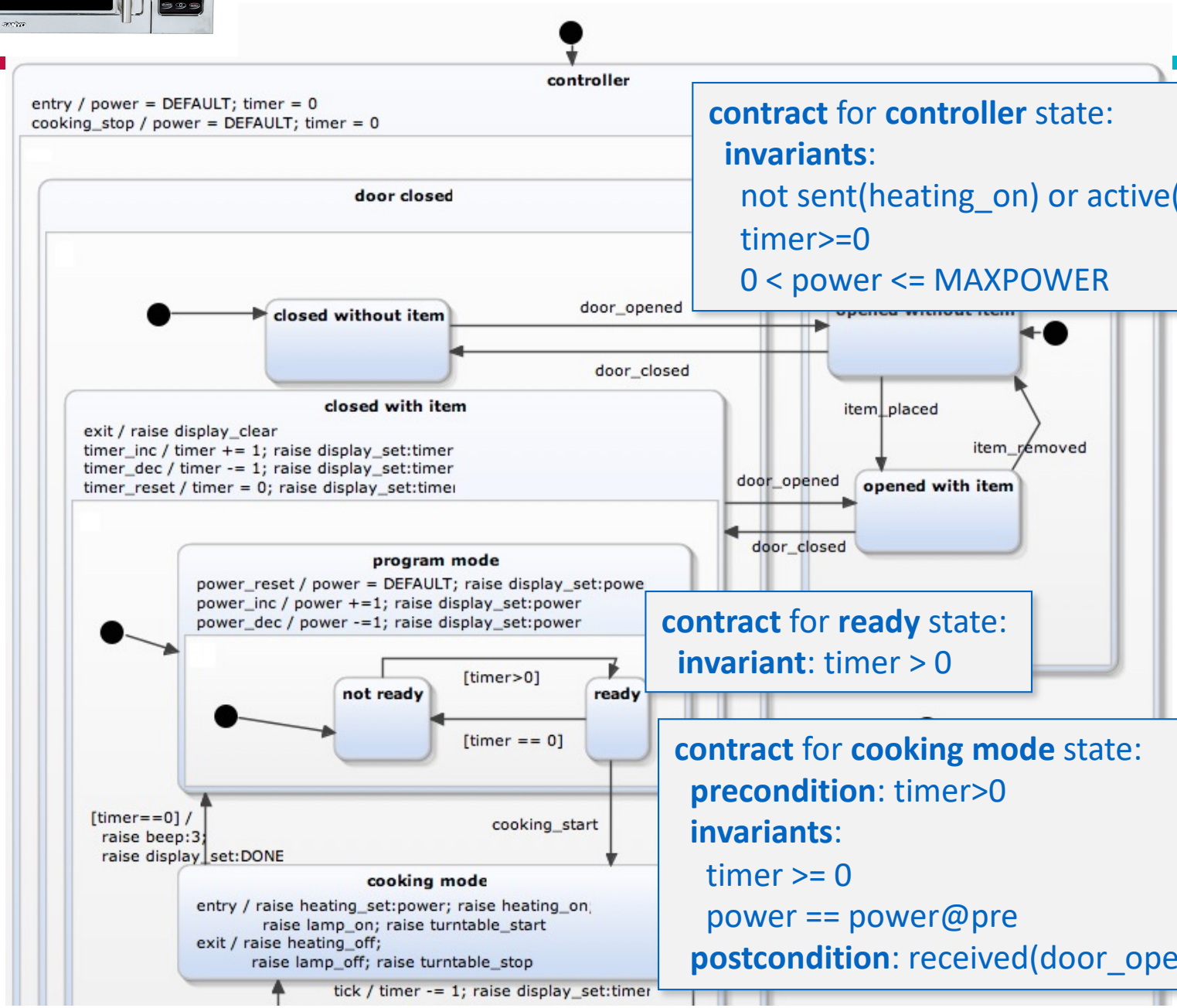
- Based on Bertrand Meyer's "*Design by Contract*" in 1986
- Add precise and dynamically verifiable specifications to executable code components (e.g., methods, functions, classes) or model components (e.g., states, transitions)
- The code (or model) should respect a *contract* composed of
  - *preconditions*
  - *postconditions*
  - *invariants*

Contract-driven development

Example (from https://www.eiffel.com/values/design-by-contract/ introduction)

```
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



contract for **controller** state:
invariants:
not sent(heating_on) or active(cooking mode)
timer>=0
0 < power <= MAXPOWER

contract for **ready** state:
invariant: timer > 0

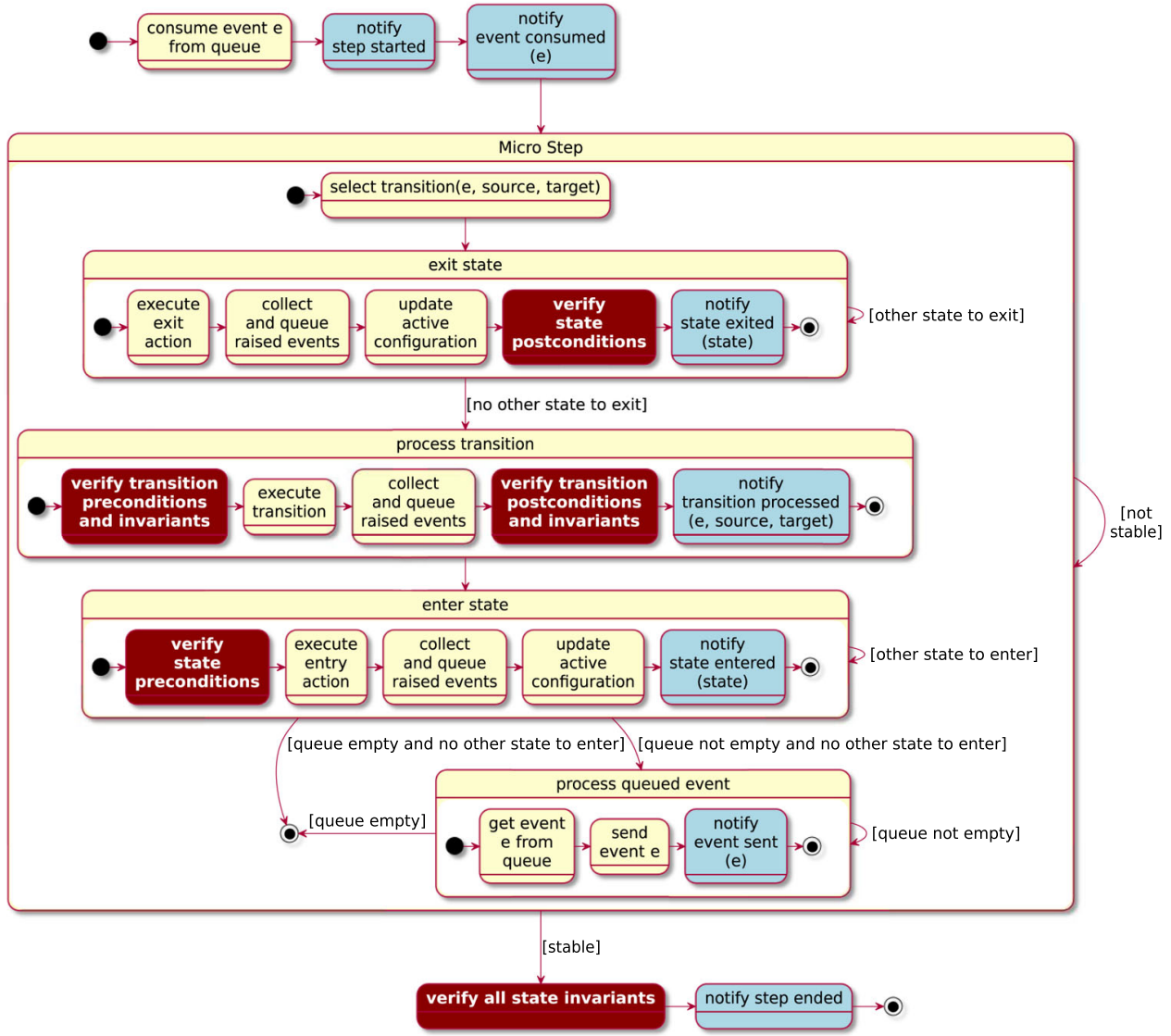
contract for **cooking mode** state:
precondition: timer>0
invariants:
timer >= 0
power == power@pre
postcondition: received(door_opened) or timer==0

Contract-driven modelling

```
statechart:
name: Microwave controller with contracts
preamble: | # as on earlier slide
root state:
  name: controller
  initial: door closed
  on entry: |
    power = POWER_DEFAULT
    timer = 0
  # ...some stuff removed here...
  contract:
    - always: not sent('heating_on') or active('cooking mode')
    - always: timer >= 0
    - always: 0 <= power <= MAXPOWER
  states:
  - name: door closed
    initial: closed with item
    states:
    - name: closed with item
      states:
      - name: cooking mode
        # ... some stuff removed here...
        contract:
        - before: timer > 0
        - always: timer >= 0
        - always: power == __old__.power or received('cooking_stop')
        - after: received('door_opened') or timer == 0
```

SISMIC
Macrostep semantics revisited

with support for runtime monitoring of contracts



SISMIC
Macrostep semantics revisited

```
interpreter.queue('door_opened’, 'item_placed', 'door_closed')
interpreter.queue('timer_dec', 'cooking_start', 'tick')])
interpreter.execute()
```

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']),

…

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

timer_dec [timer>0] / timer = timer - 1

power_inc [power<MAXPOWER] / power = power+1

power_dec [power>1] / power = power -1

Test-driven development

Using Python's unittest module

```
class MicrowaveTests(unittest.TestCase):
    def setUp(self):
        # code here for setting up statechart interpreter
    def test_negative_timer(self):
        self.interpreter. queue (door_opened, item_placed, door_closed, timer_dec)
        interpreter.execute()
        self.assertEqual(self.interpreter.context['timer'], 0)
```

Without guards on timer_dec event

```
test negative_timer ... FAIL
======================================
AssertionError: -1 != 0
--------------------------------------------------------------
Ran 1 tests in 0.005s
FAILED (failures=1)
```

Test-driven development

Using Python's unittest module

```
class MicrowaveTests(unittest.TestCase):
    def setUp(self):
        # code here for setting up statechart interpreter
    def test_negative_timer(self):
        self.interpreter. queue (door_opened, item_placed, door_closed, timer_dec)
        interpreter.execute()
        self.assertEqual(self.interpreter.context['timer'], 0)
```

With guards on timer_dec event

```
test negative_timer ... ok
----------------------------------------------------------------------
Ran 1 tests in 0.005s
OK
```

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)

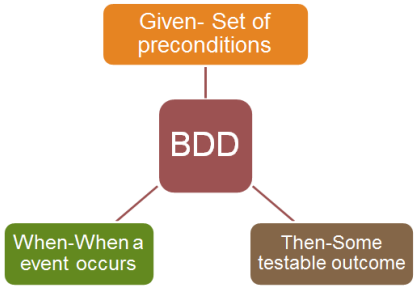
Remaining states:

cooking mode | ready

Transition coverage: 16.67%

Processed transitions:

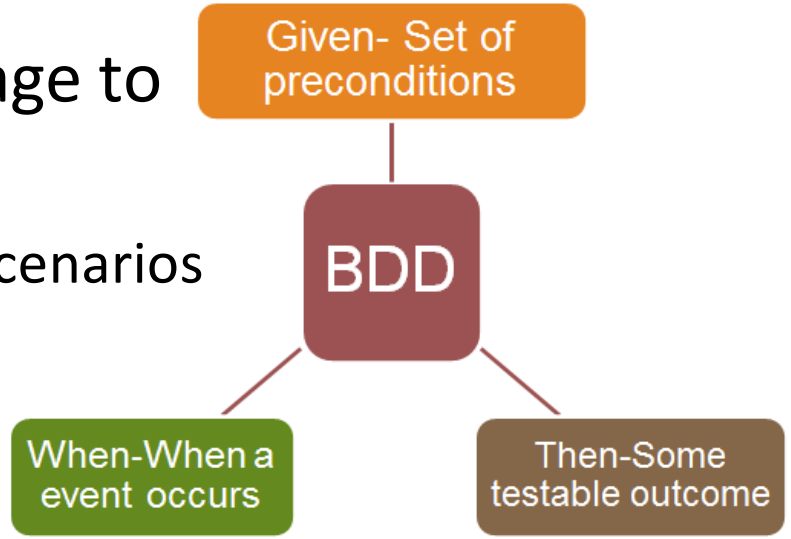
opened without item [item_placed] -> opened with item (2)
closed without item [door_opened] -> opened without item (2)
opened with item [door_closed] -> closed with item (1)

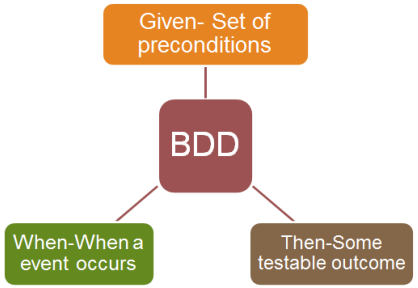


Behaviour-Driven Development

- Include customer test practices into TDD
  - Encourage collaboration between developers, QA, and non- technical stakeholders (domain experts, project managers, users)
  - Reduce technical gap between developers and other project stakeholders

- Use a domain-specific textual language to specify how the code should behave
  - By defining feature specifications and scenarios
  - Using Gherkin language
  - Supported by Cucumber framework



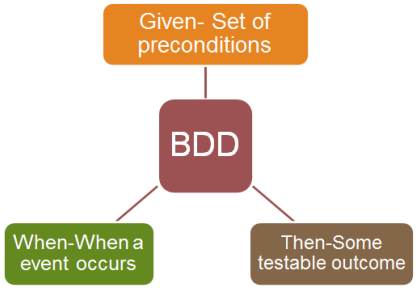


Behaviour-Driven Development

Example
(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



Behaviour-Driven Modeling

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

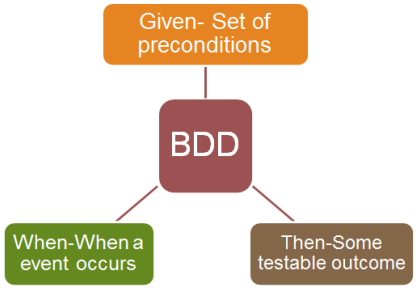
When I close the door

Then heating should not tur

Microwave scenarios
Described in (structured) textual language, without any knowledge of the statechart's internals.

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



Behaviour-Driven Modeling

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

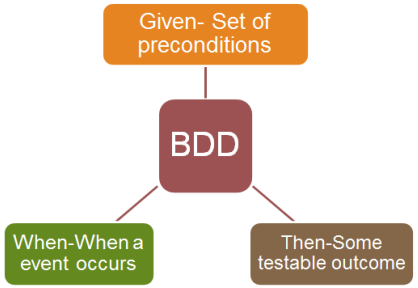
And I send event item_placed

When I send event door_closed

Then event heating_on should not be fired

Intermediate (statechart-specific) scenarios

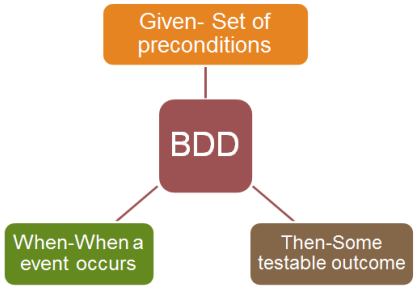
Automatically generated from high-level scenarios through user-defined **mapping**.



Behaviour-Driven Modeling

Mapping specification

```
from sismic.bdd import map_action, map_assertion
map_action('I open the door', 'I send event door_opened')
map_action('I close the door', 'I send event door_closed')
map_action('I place an item in the oven', 'I send event item_placed')
map_action('I press increase timer button {time} times', 'I repeat "I send event timer_inc" {time} times')
map_action('I press increase power button', 'I send event power_inc')
map_action('I press start button', 'I send event cooking_start')
map_action('I press stop button', 'I send event cooking_stop')
map_action('{tick} seconds elapsed', 'I repeat "I send event timer_tick" {tick} times')
map_assertion('Heating turns on', 'Event heating_on is fired')
map_assertion('Heating does not turn on', 'Event heating_on is not fired'
map_assertion('heating turns off', 'Event heating_off is fired')
map_assertion('lamp turns on', 'Event lamp_switch_on is fired')
map_assertion('lamp turns off', 'Event lamp_switch_off is fired')
```



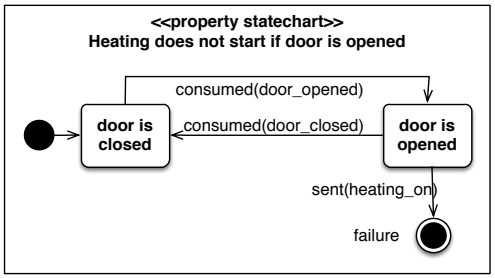
SISMIC
Behaviour-Driven Modeling

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)
```



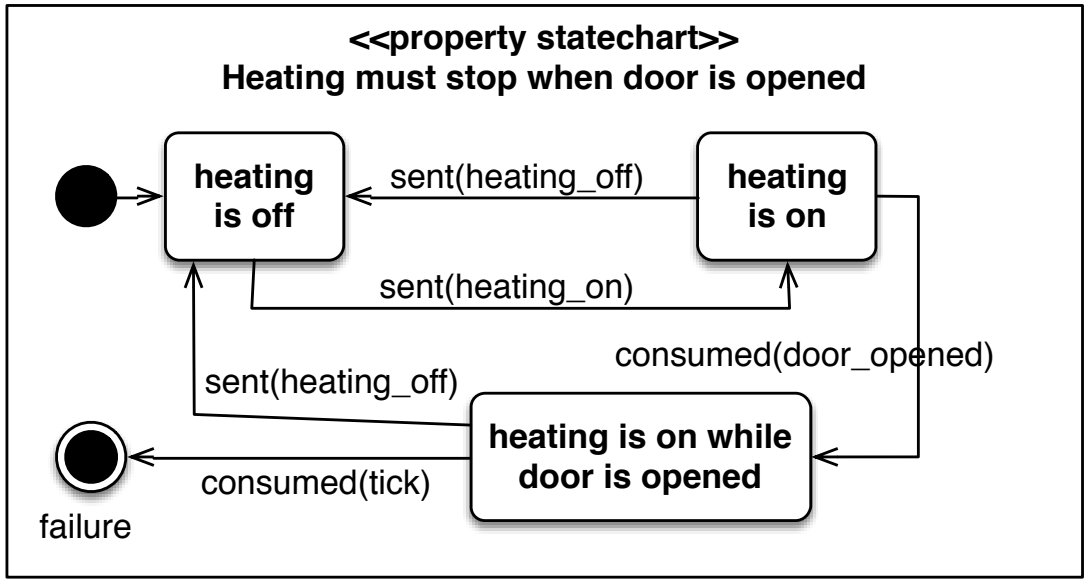
SISMIC
Property statecharts

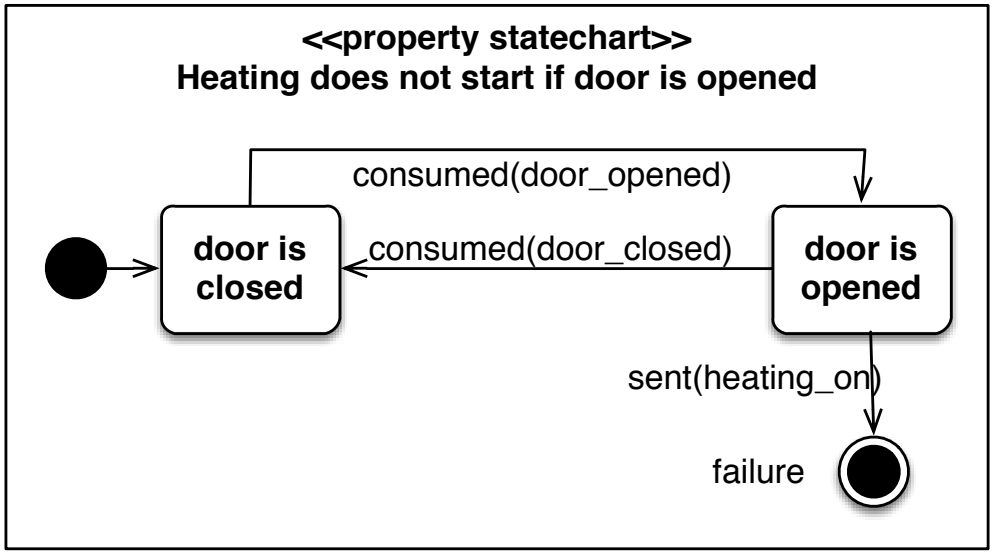
Define and verify behavioural properties by

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

SISMIC
Property statecharts

Examples





SISMIC
Communicating statecharts

- Statecharts can *communicate* with other statecharts or external components (e.g. a user interface) by sending/receiving events
- This is realised by *binding* their statechart interpreters

SISMIC
Communicating statecharts

Example: Events sent by buttons are propagated to elevator

```
buttons = Interpreter(import_from_yaml(open('buttons.yaml')))
elevator = Interpreter(import_from_yaml(open('elevator.yaml')))
buttons.bind(elevator)

buttons.queue(Event('floor_2_pushed'))
buttons.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
```

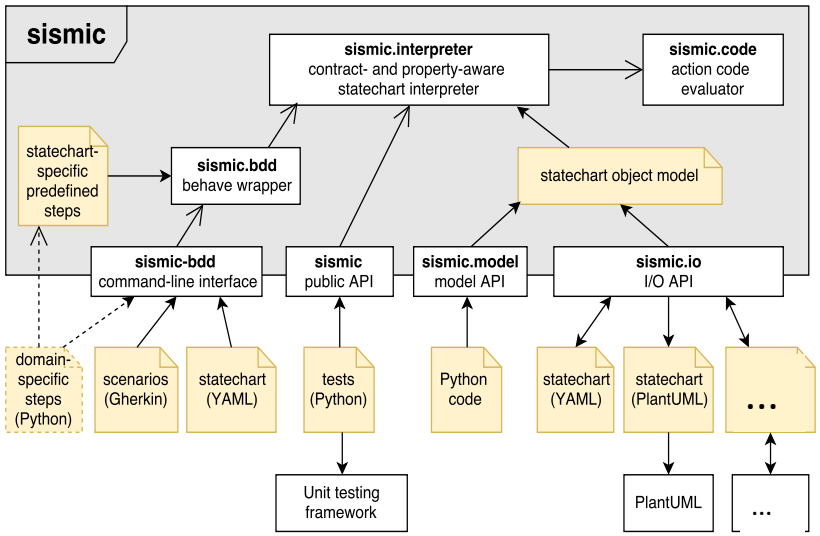
SISMIC
Other features

- Use other semantic variants of statecharts by overriding private methods of Interpreter
  - E.g. outer-first instead of inner-first semantics; changing priority of events
- Different ways of dealing with time
  - Real time versus simulated time

Future Work

- Model quality assessment (model smells) and improvement (model refactoring)
  - E.g. splitting up a complex statechart into multiple interacting statecharts
- More advanced testing techniques
  - Automatic generation of contracts based on scenario specifications
  - Automatic generation of tests based on contract specifications
  - Mutation testing
  - …
- Explore (dynamic?) model checking techniques
  - E.g., based on LTL formulas, using Dwyer's specification patterns, others?
- Model 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
- Model composition and scalability
- Semantic variation
  - Detecting if statechart is compatible with alternative statechart semantics

Conclusion / Summary

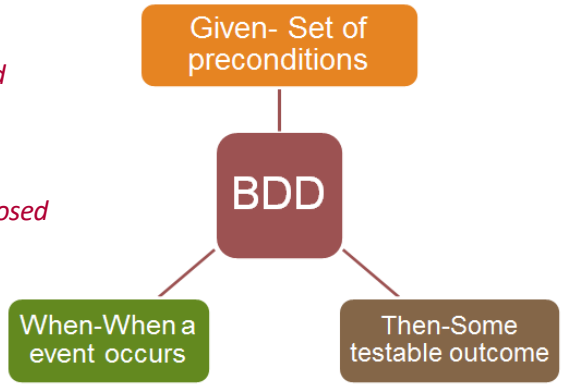


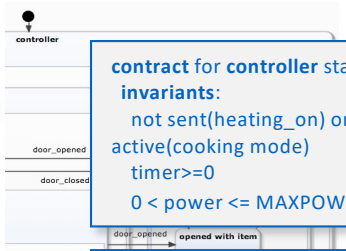
Feature: *No heating if door is opened*

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

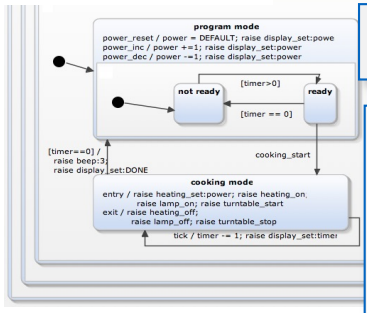
Scenario: *No heating when item is placed*
Given I open the door
When I place an item
Then heating should not turn on

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





contract for **controller** state:
invariants:
not sent(heating_on) or active(cooking mode)
timer>=0
0 < power <= MAXPOWER



contract for **ready** state:
invariant: timer > 0

contract for **cooking mode** state:
precondition: timer>0
invariants:
timer >= 0
power == power@pre
postcondition:
received(door_opened) or timer==0