Developing Reactive Systems using Statecharts

Modelling of Software-Intensive Systems

Simon Van Mierlo
University of Antwerp
Belgium
simon.vanmierlo@uantwerpen.be

Hans Vangheluwe
University of Antwerp
Belgium
hans.vangheluwe@uantwerpen.be

Axel Terfloth
itemis AG
Germany
terfloth@itemis.de

Introduction

Reactive Systems

- Com **our**
- In contrast to *transformational* systems, which take input and, eventually, produce output

Modelling Reactive Systems

- Interaction with the environment: reactive to *events*
- Autonomous behaviour: *timeouts* + *spontaneous* transitions
- System behaviour: *modes* (hierarchical) + *concurrent* units
- Use programming language + threads and tasks (OS)?

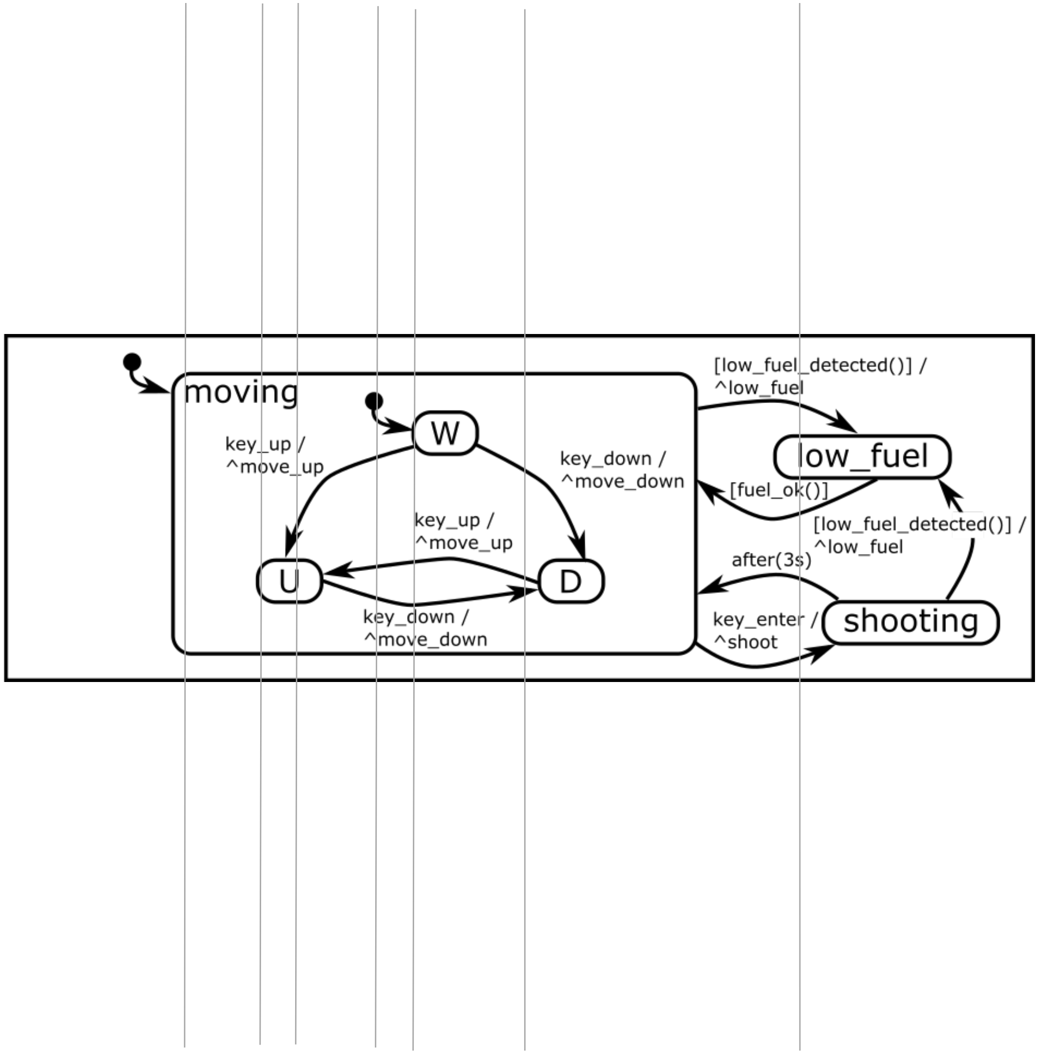
"Nontrivial software written with threads, semaphores, and mutexes are incomprehensible to humans"

Programming language (and OS) is too low-level
-> most appropriate formalism: "what" vs. "how"

E. A. Lee, "The problem with threads," in *Computer*, vol. 39, no. 5, pp. 33-42, May 2006.

Discrete-Event Abstraction

behavioural
model



moving

W

U

D

low_fuel

shooting

key_up /
^move_up

key_down /
^move_down

key_up /
^move_up

key_down /
^move_down

[low_fuel_detected()] /
^low_fuel

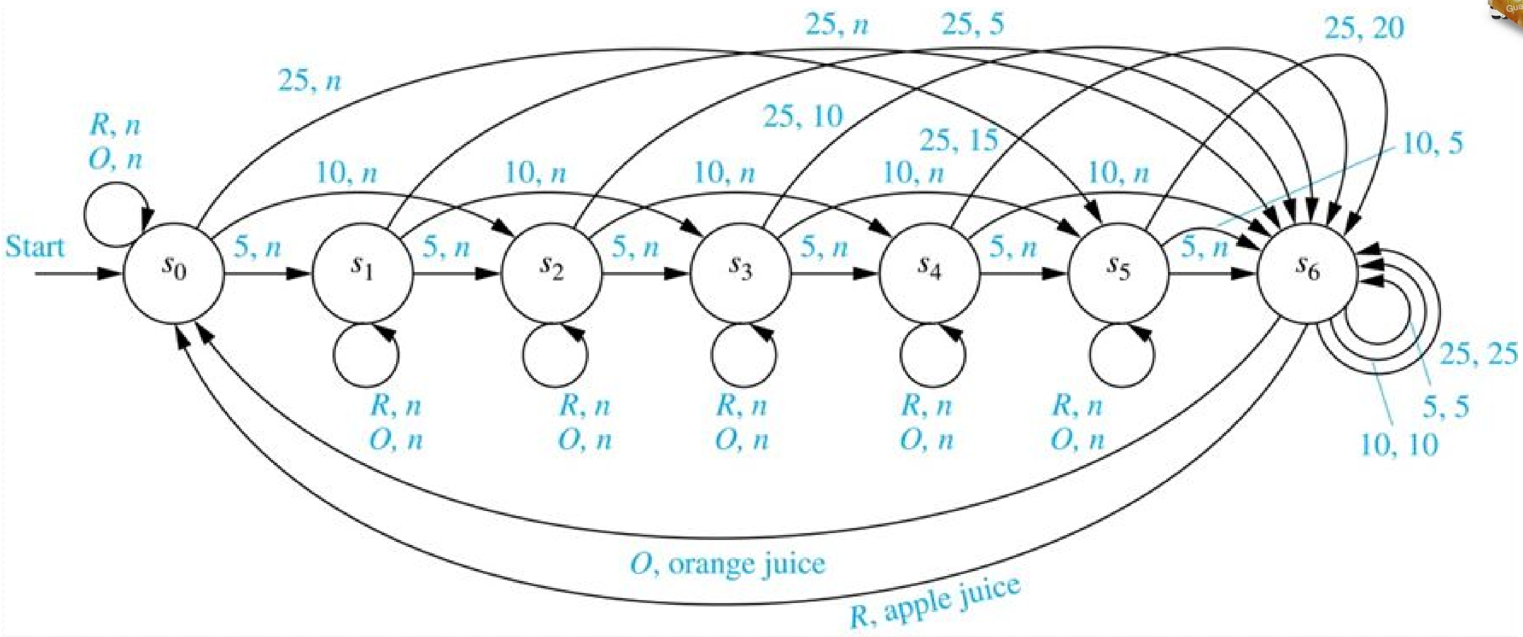
[fuel_ok()]

[low_fuel_detected()] /
^low_fuel

after(3s)

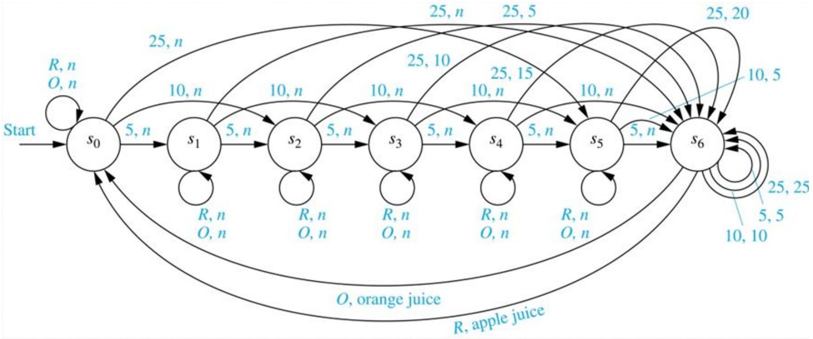
key_enter /
^shoot

State Diagrams



- All states are explicitly represented (unlike Petrinets, for example)
- Flat representation (no hierarchy)
- Does not scale well: becomes too large too quickly to be usable (by humans)

Alternative Representation: Parnas Tables

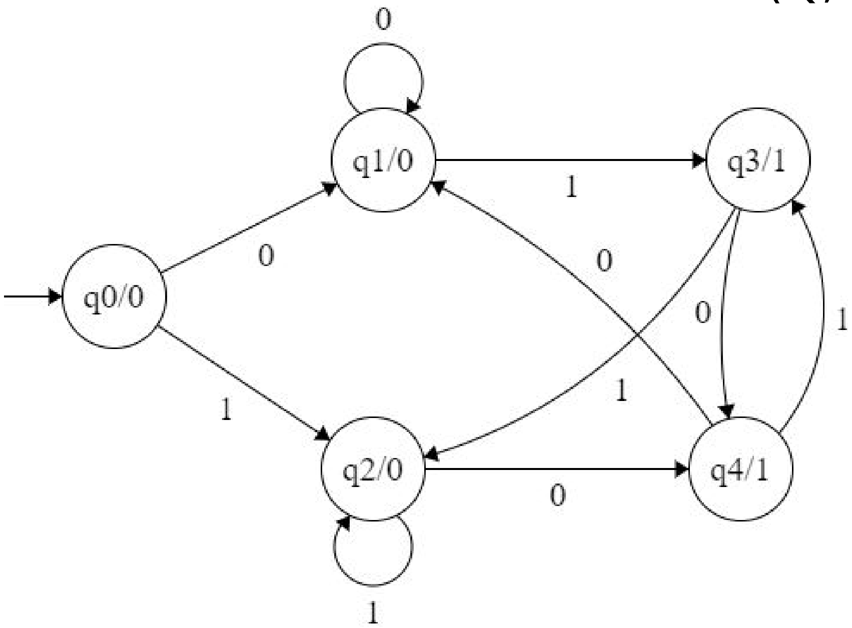


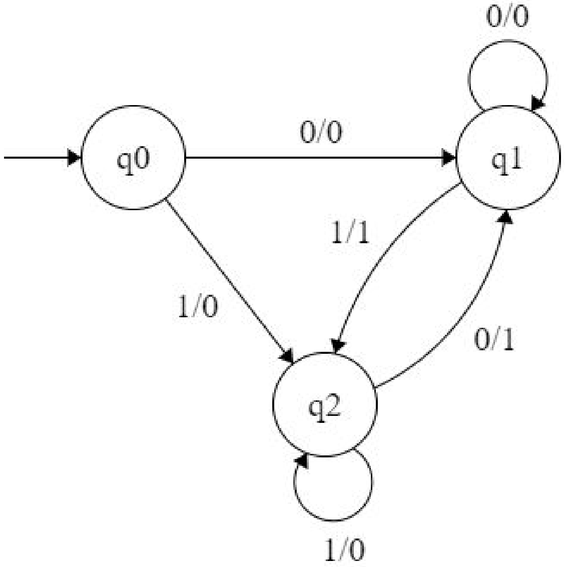
event/state	s_0	s_1	s_2	s_3	s_4	s_5	s_6
5	s_1, n	s_2, n	s_3, n	s_4, n	s_5, n	s_6, n	s_6, 5
10	s_2, n	s_3, n	s_4, n	s_5, n	s_6, n	s_6, 5	s_6, 10
25	s_5, n	s_6, n	s_6, 5	s_6, 10	S_6, 15	s_6, 20	s_6, 25
O	s_0, n	s_1, n	s_2, n	s_3, n	s_4, n	s_5, n	s_0, orange juice
R	s_0, n	s_1, n	s_2, n	s_3, n	s_4, n	s_5, n	s_0, apple juice

https://cs.uwaterloo.ca/~jmatlee/Talks/Parnas01.pdf

Mealy and Moore Machines

FSA: (Q, q0, ∑, O,   ,   )





Moore Machines

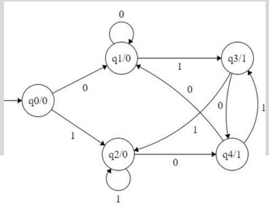
- Output only depends on current state.
  : Q   O
- Input stream: 10
  Output stream: 00

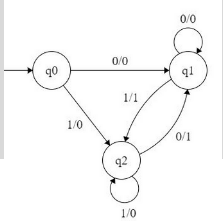
Mealy Machines

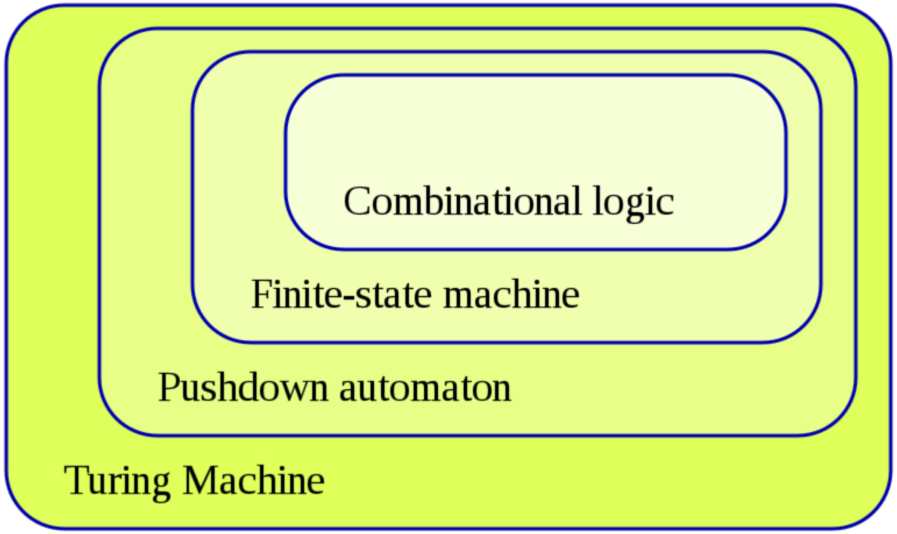
- Output depends on current state and on current input.
  : Q x ∑   O
- Input stream: 10
  Output stream: 01

https://www.geeksforgeeks.org/mealy-and-moore-machines/

FSAs: Expressiveness







Combinational logic

Finite-state machine

Pushdown automaton

Turing Machine

- Can be made Turing-complete

  data memory, control flow, branching

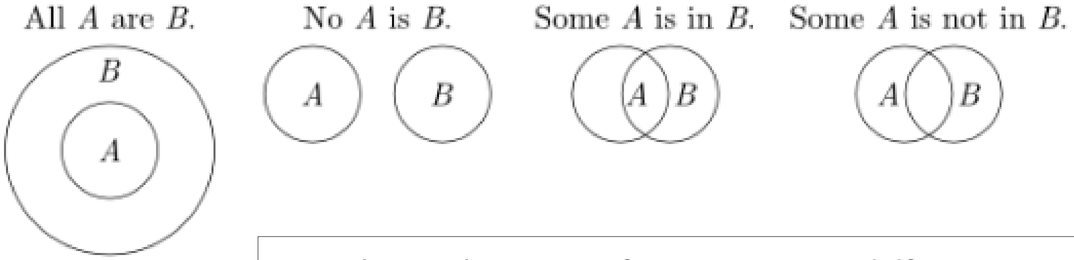
- Extend FSAs

  borrow semantics from Mealy and Moore machines

https://en.wikipedia.org/wiki/Finite-state_machine

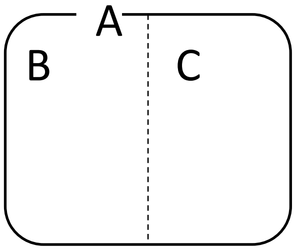
Higraphs

Euler Diagrams



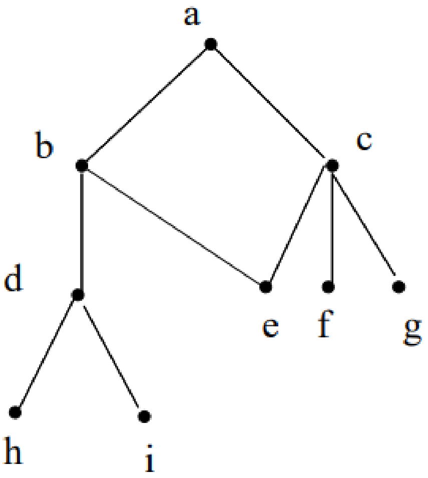
topological notions for set union, difference, intersection

Unordered Cartesian Product

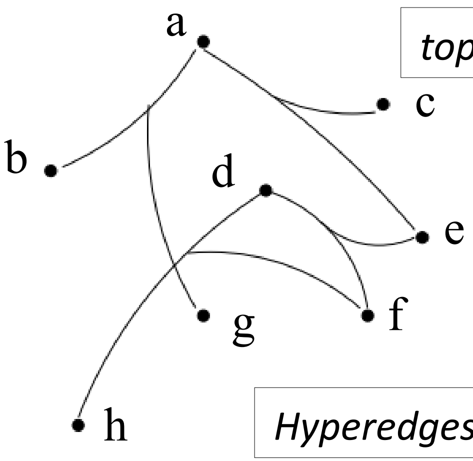


A = B   C

Hypergraphs



a graph



a hypergraph

topological notion (syntax): connectedness

Hyperedges: 2^X (undirected), $2^X \times 2^X$ (directed).

$X = \{a, b, \ldots, h\}$

David Harel. On Visual Formalisms. Communications of the ACM. Volume 31, No. 5. 1988. pp. 514 - 530.

Higraphs

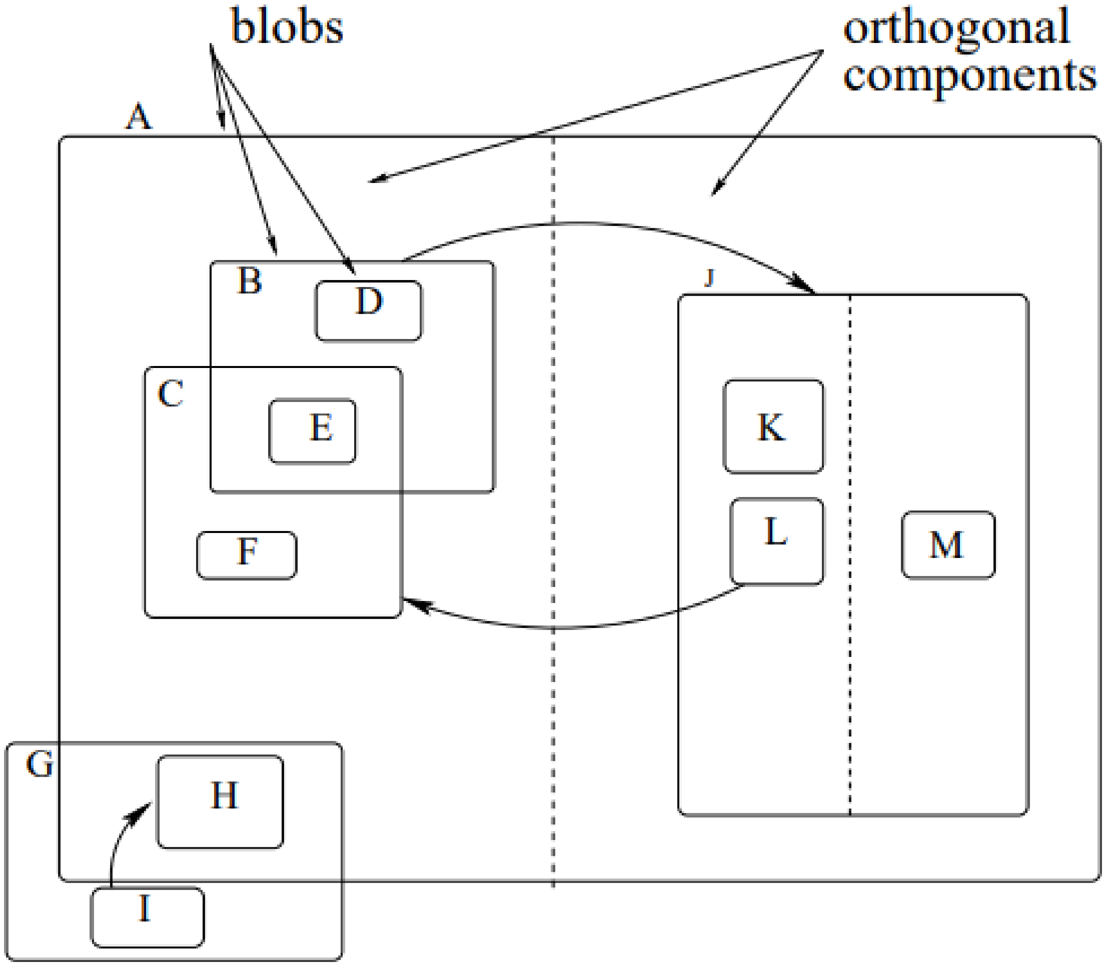
Euler Diagrams

+

Hypergraphs

+

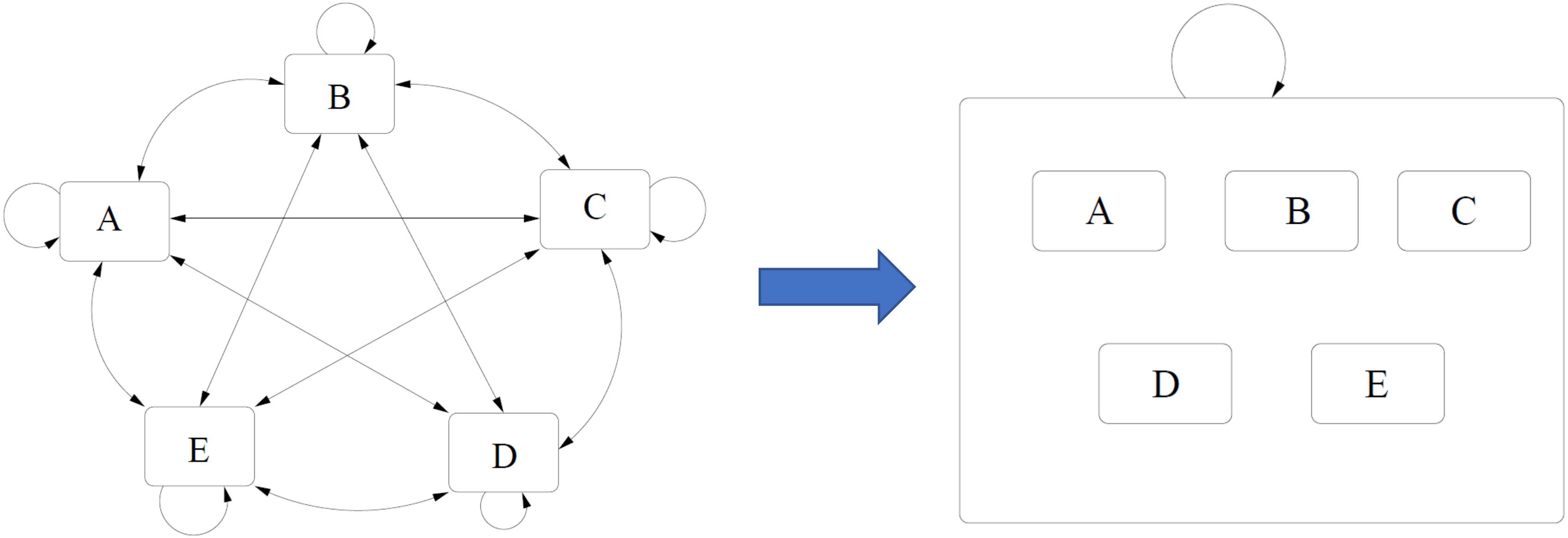
Unordered Cartesian Product



David Harel. On Visual Formalisms. Communications of the ACM. Volume 31, No. 5. 1988. pp. 514 - 530.

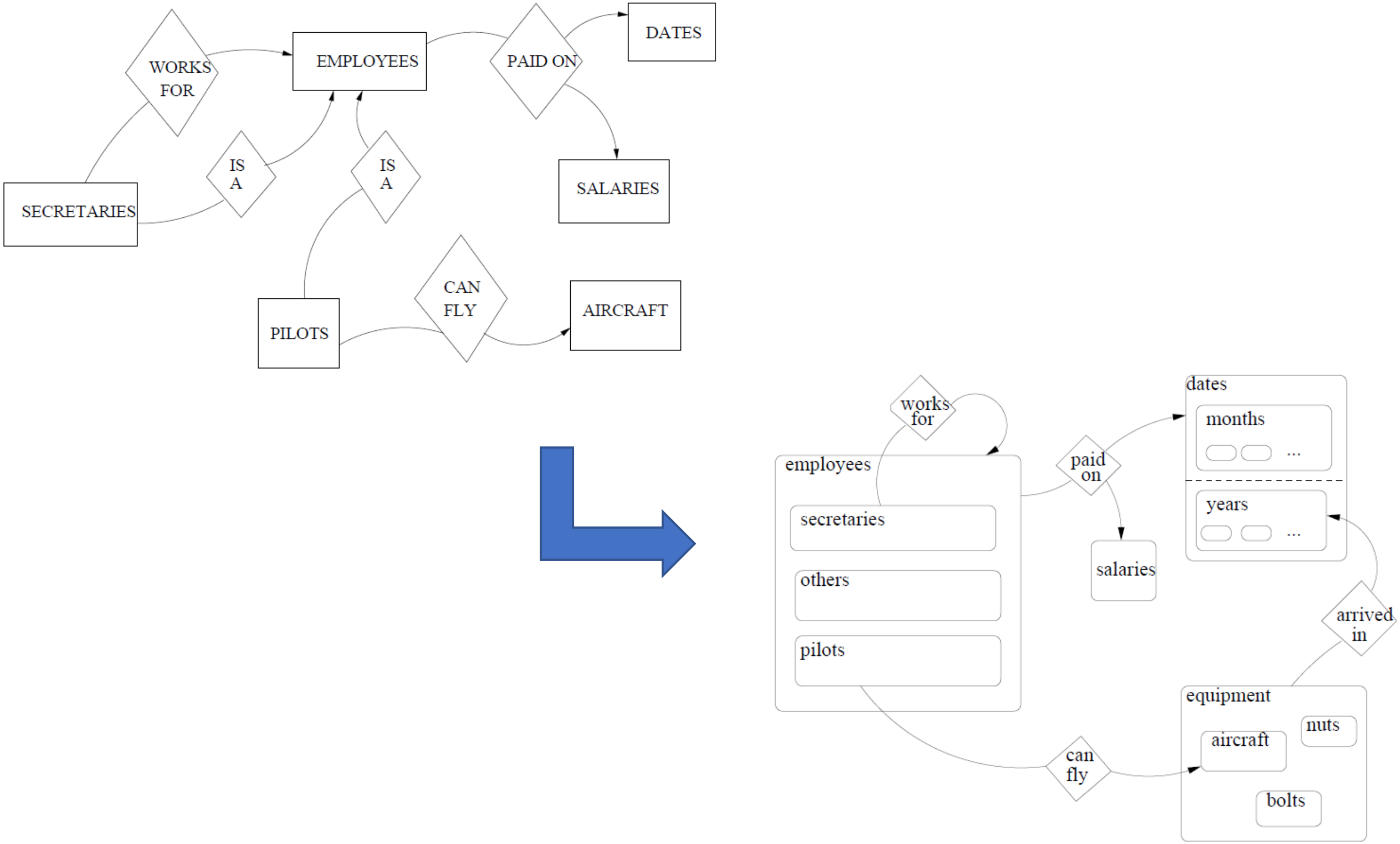
Higraph: Examples

- Clique

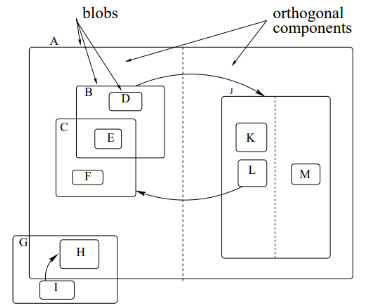


Higraphs: Examples

- ER-Diagrams



Higraphs: Formal Definition



- A higraph *H* is a quadruple

  H = (B, E,

- B is a finite set of all unique *blobs*
- E is a set of *hyperedges*

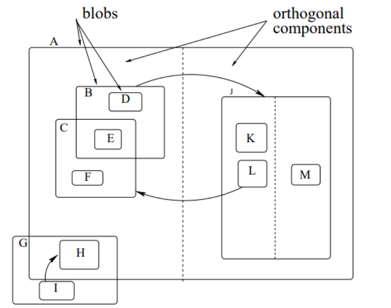
  2^B x 2^B

- The subblob function

  : B 2^B

$$\sigma^{i+1}(x) = \bigcup_{y \in \sigma^i(x)} \sigma(y) \quad \sigma^{+}(x) = \bigcup_{i=1} \sigma^i(x)$$

Higraphs: Formal Definition



- Subblobs relation cycle-free

x (x)

- The partitioning function π associates an *equivalence relationship* with x

$B \quad 2^{B \times B}$

- Equivalence classes π_i are orthogonal components of x

$$\pi_1(x), \pi_2(x), \ldots, \pi_{kx}(x)$$

- $k_x = 1$ means a single orthogonal component
- Blobs in different orthogonal components of x are disjoint

$y,z \quad (x) : \quad {}^{+}(y) \quad {}^{+}(z) =$

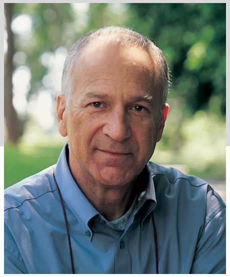
Higraphs Applications

- Apply syntactic constructs to an existing modelling language.
- Add specific meaning to these constructs.
- Examples:
  - E-R diagrams
  - Dataflow/Activity Diagrams
  - Inheritance
  - **Statecharts**

Statecharts

- Visual (topological, not geometric) formalism
- Precisely defined syntax and semantics
- Many uses:
  - Documentation (for human communication)
  - ~~Analysis (of behavioural prop~~erties)
  - Simulation
  - Code synthesis
  - … and derived, such as testing, optimization, …

Statecharts History



- Introduced by David Harel in 1987
- Notation based on higraphs = hypergraphs + Euler diagrams + unordered Cartesian product
- Semantics extends deterministic finite state automata with:
  - Depth (Hierarchy)
  - Orthogonality
  - Broadcast Communication
  - Time
  - History
  - Syntactic sugar, such as enter/exit actions

David Harel, Statecharts: a visual formalism for complex systems, Science of Computer Programming, Volume 8, Issue 3, 1987, pages 231-274

Statecharts History

- Incorporated in UML: State Machines (1995)
- More recent: xUML for semantics of UML subset (2002)
- W3 Recommendation: State Chart XML (SCXML) (2015)

  https://www.w3.org/TR/scxml/
- Standard: Precise Semantics for State Machines (2019)

  https://www.omg.org/spec/PSSM/

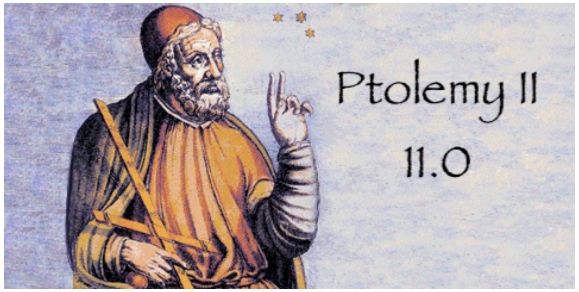
Statechart (Variants) Tools

STATEMATE: A Working Environment for the Development of Complex Reactive Systems

Rational® software

https://www.ibm.com/us-en/marketplace/systems-design-rhapsody

https://www.mathworks.com/products/stateflow.html



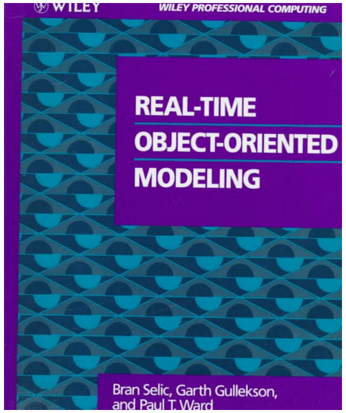
https://ptolemy.berkeley.edu/ptolemyII/ptII11.0/index.htm

YAKINDU STATECHART TOOLS

https://www.itemis.com/en/yakindu/state-machine/

PAPYRUS REALTIME

https://www.eclipse.org/papyrus-rt/



WILEY

WILEY PROFESSIONAL COMPUTING

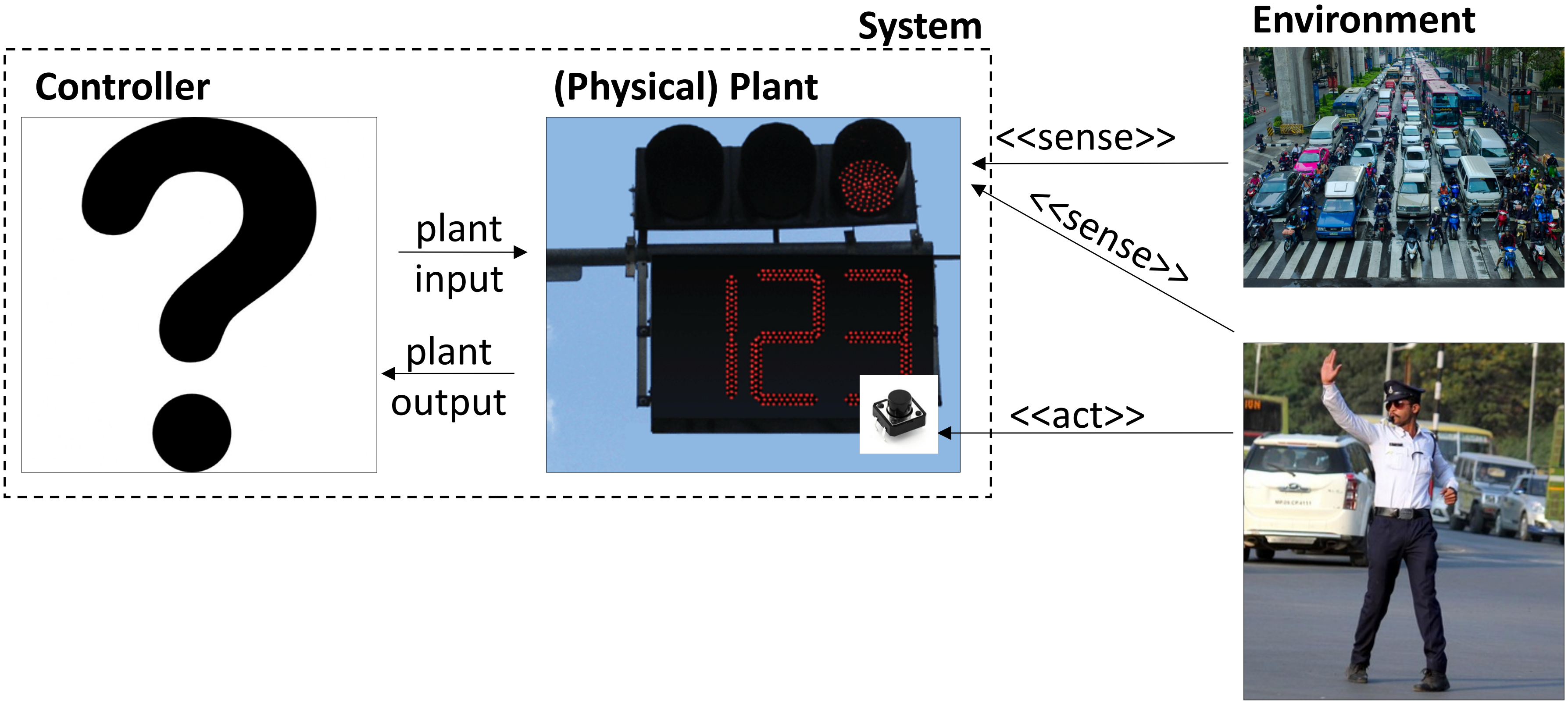
REAL-TIME OBJECT-ORIENTED MODELING

Bran Selic, Garth Gullekson, and Paul T. Ward

etrice.

https://www.eclipse.org/etrice/

Running Example



System

Environment

Controller

(Physical) Plant

plant input

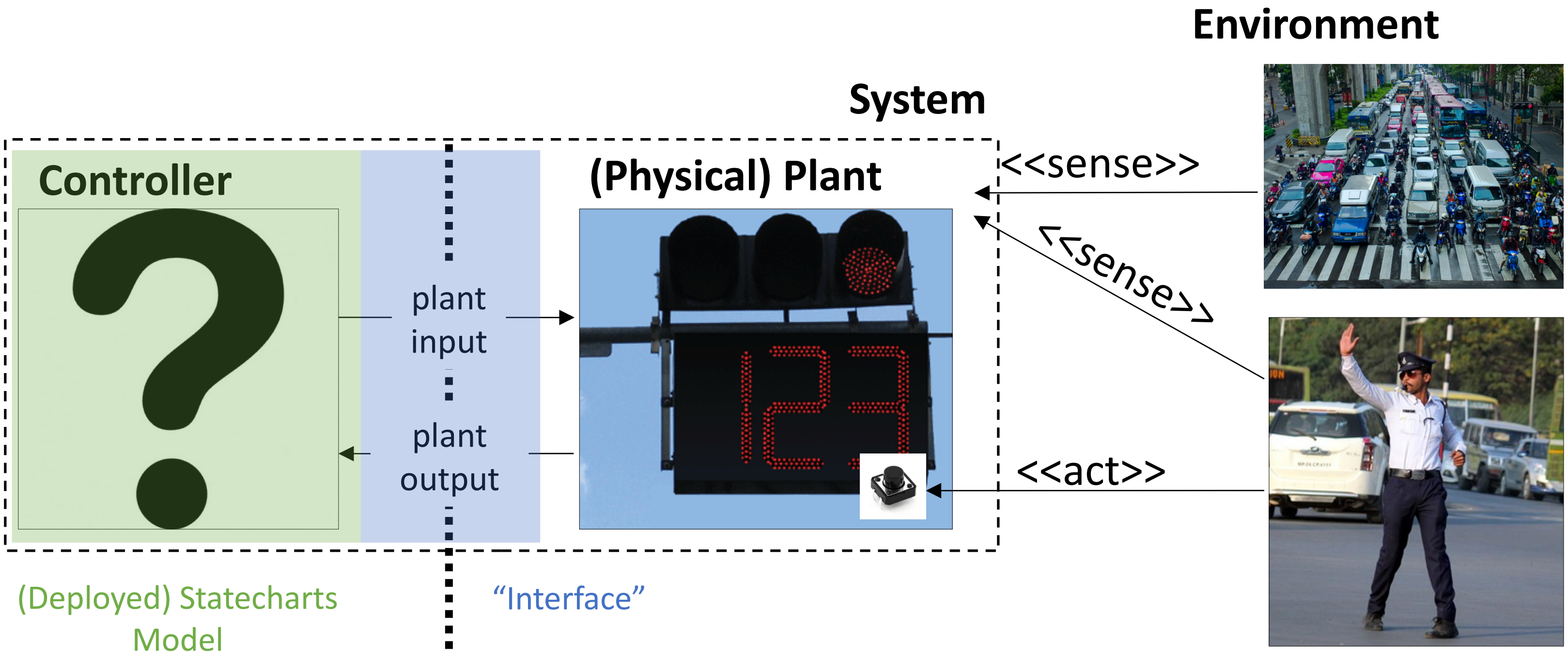
plant output

<<sense>>

<<sense>>

<<act>>

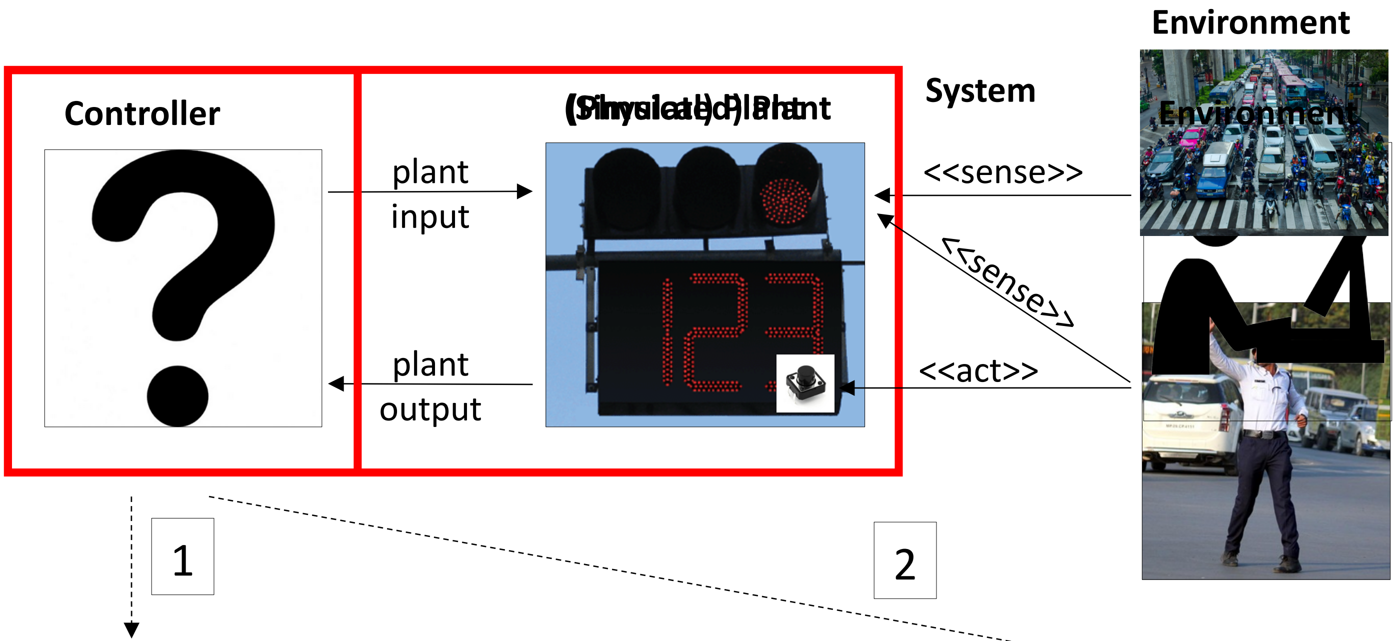
What are we developing?

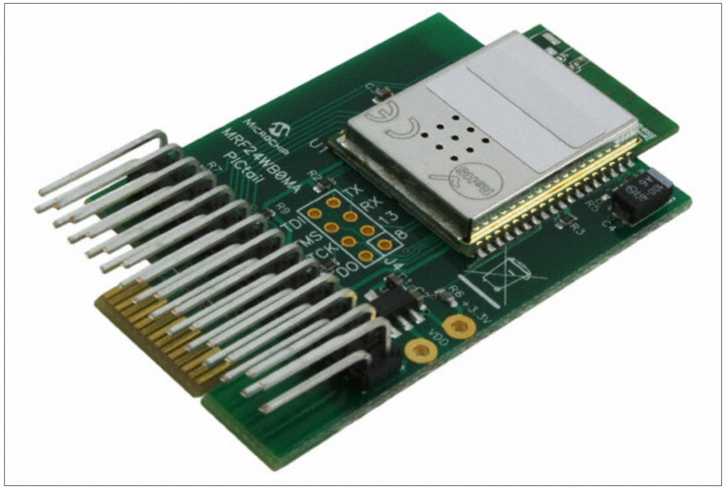


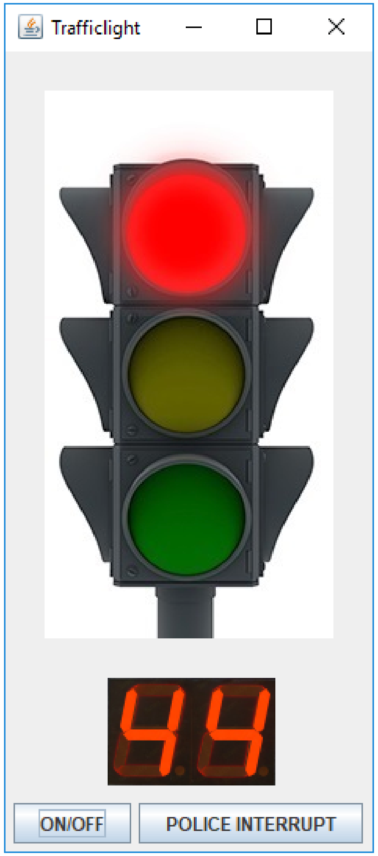
- Autonomous (timed) behaviour
- Interrupt logic
- Orthogonal (traffic light/timer) behaviour

- Turn on/off traffic lights (red/green/yellow)
- Display counter value (three-digit)
- Change counter colour (red/green)
- Sense button presses

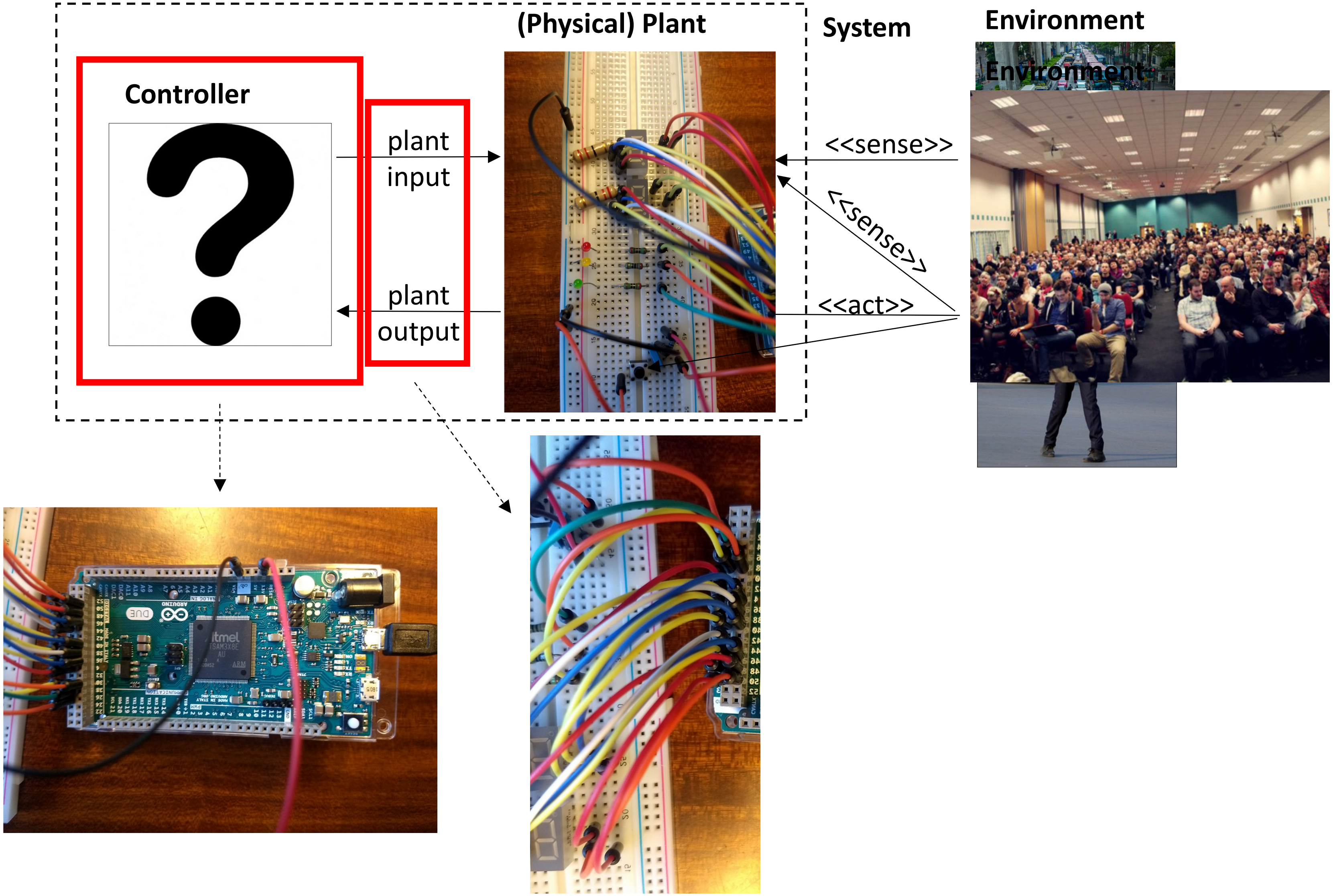
Deployment (Simulation)







Deployment (Hardware)



(Physical) Plant

System

Environment

Environment

Controller

plant input

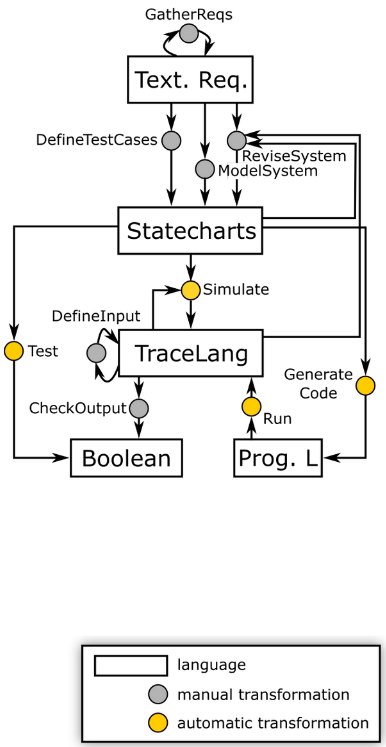
plant output

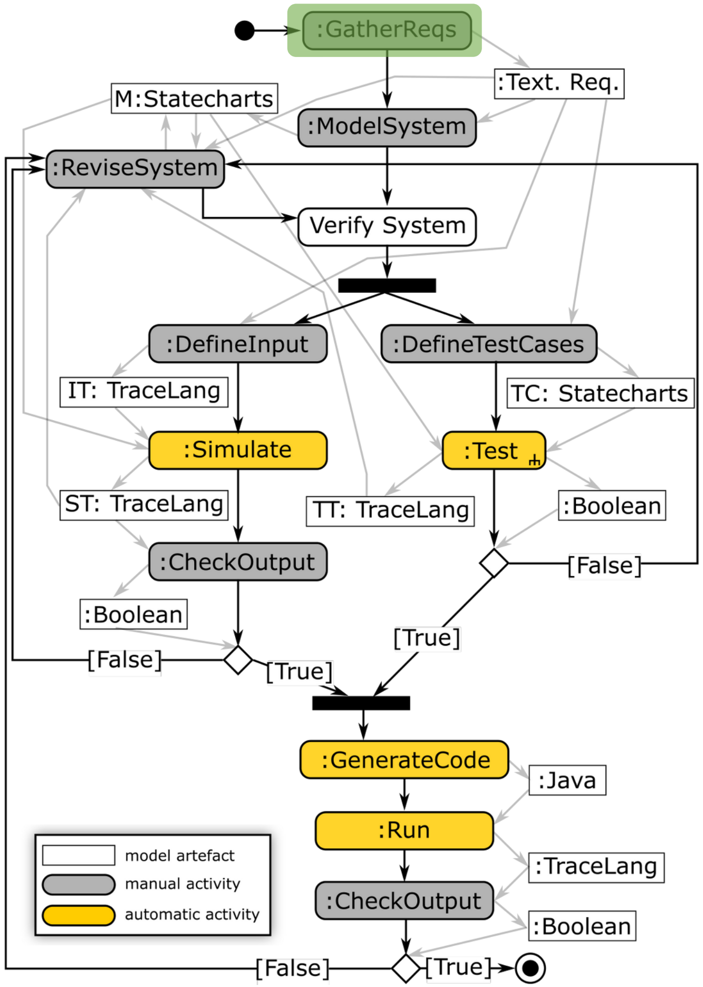
<<sense>>

<<sense>>

<<act>>

Workflow

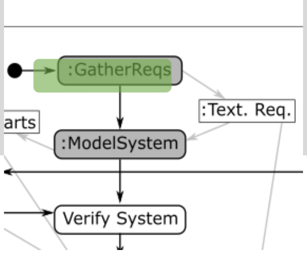


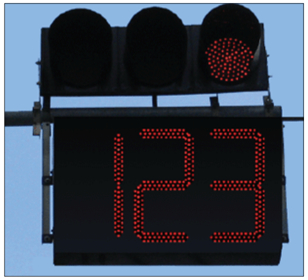


Hans Vangheluwe and Ghislain C. Vansteenkiste. A multi-paradigm modeling and simulation methodology: Formalisms and languages. In European Simulation Symposium (ESS), pages 168-172. Society for Computer Simulation International (SCS), October 1996. Genoa, Italy.

Levi Lúcio, Sadaf Mustafiz, Joachim Denil, Hans Vangheluwe, Maris Jukss. FTG+PM: An Integrated Framework for Investigating Model Transformation Chains. System Design Languages Forum (SDL) 2013, Montreal, Quebec. LNCS Volume 7916, pp 182-202, 2013.

Requirements





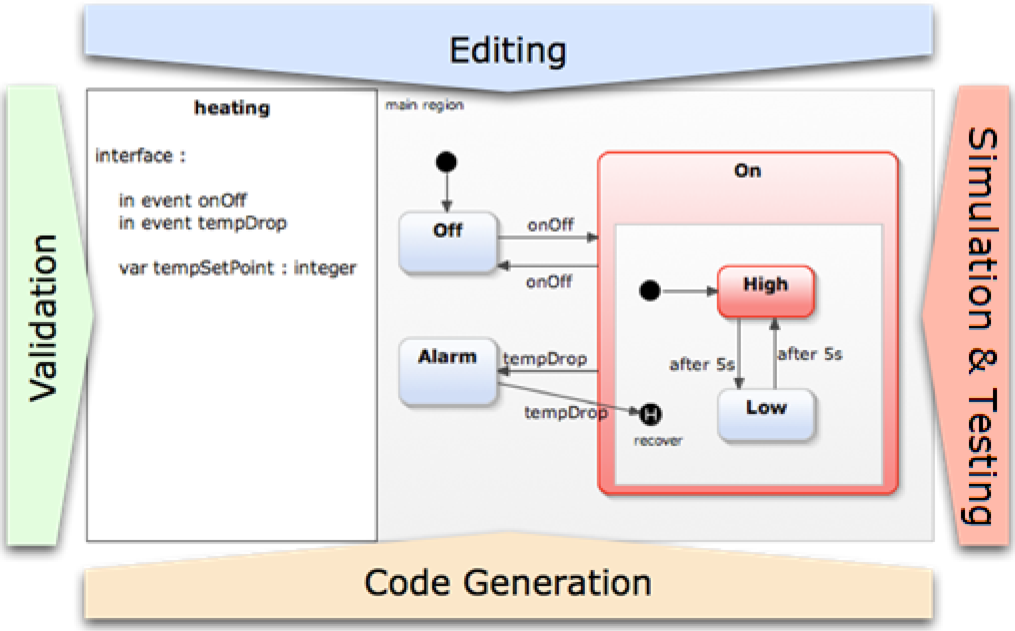
- R1: three differently coloured lights: red, green, yellow
- R2: at most one light is on at any point in time
- R3: at system start-up, the red light is on
- R4: cycles through red on, green on, and yellow on
- R5: red is on for 60s, green is on for 55s, yellow is on for 5s
- R6: time periods of different phases are configurable.
- R7: police can interrupt autonomous operation
  - Result = blinking yellow light (on -> 1s, off -> 1s)
- R8: police can resume an interrupted traffic light
  - Result = light which was on at time of interrupt is turned on again
- R9: traffic light can be switched on and off and restores its state
- R10: a timer displays the remaining time while the light is red or green; this timer decreases and displays its value every second. The colour of the timer reflects the colour of the traffic light.

YAKINDU Statechart Tools

Statecharts made easy…

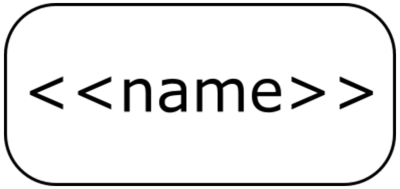
What are YAKINDU Statechart Tools?

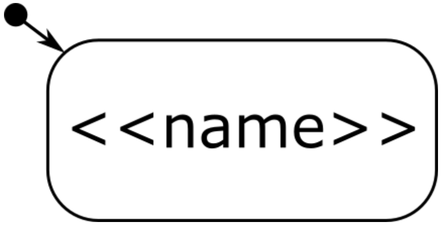
YAKINDU Statechart Tools provides an **integrated modeling environment** for the specification and development of **reactive, event-driven systems** based on the concept of statecharts.



The Statecharts Language

States





being **in** a state

= state <<*name*>> is **active**

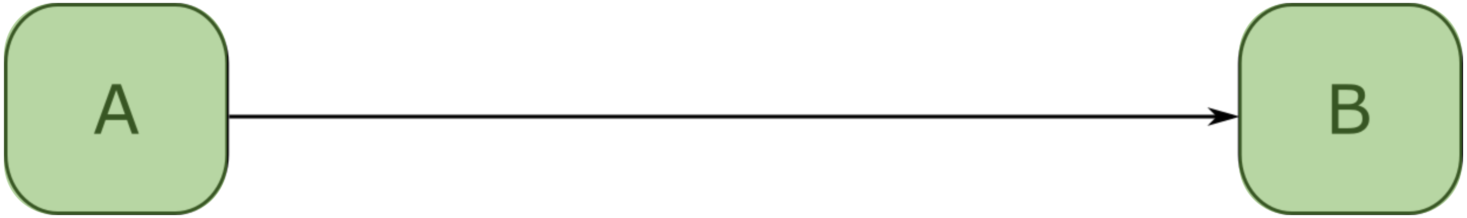
= the system is in **configuration** <<*name*>>

initial state

exactly one per model

“entry point”

Transitions

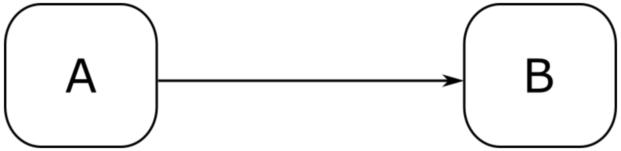


- Model the **dynamics** of the system:
  - *when*
    - the system is **in state A** and
    - the **event** is **received**
  - *then*
    1. **output_action** is evaluated and
    2. the new **active state** becomes B

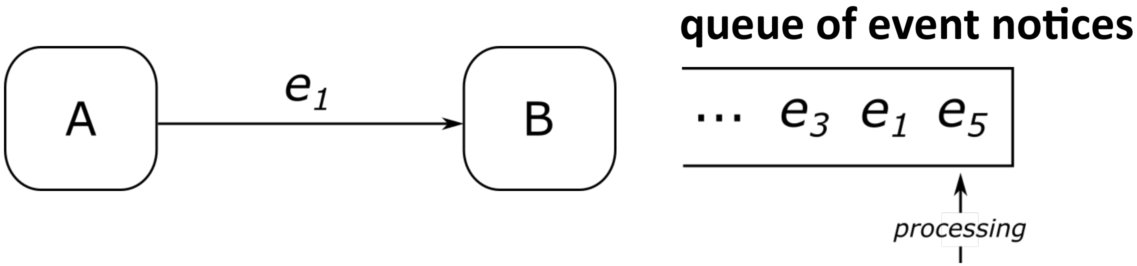
Transitions: Events

event(in_params) / output_action(out_params)

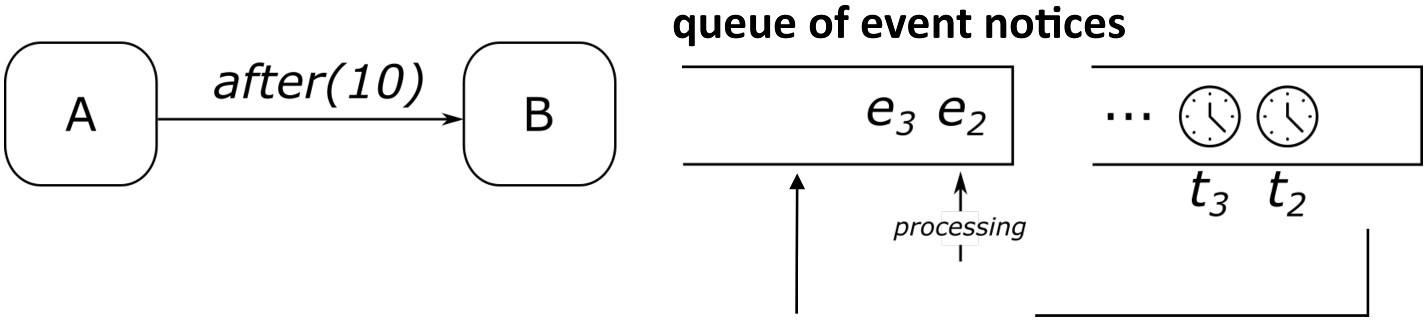
- Spontaneous



- Input Event

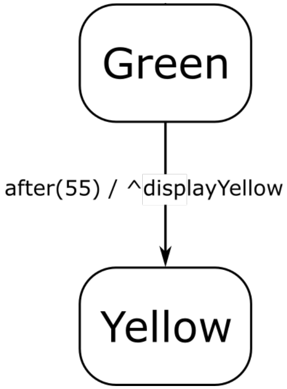


- After Event



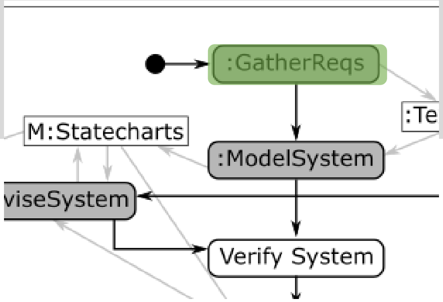
Transitions: Raising Output Events

event(in_params) / output_action(out_params)



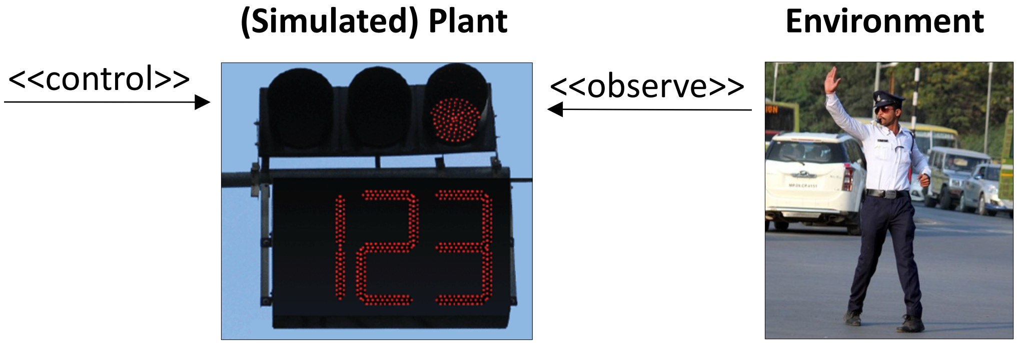
Syntax for output action:
^*output_event*
means “raise the event *output_event* (to the environment)”

Exercise 1 - Requirements

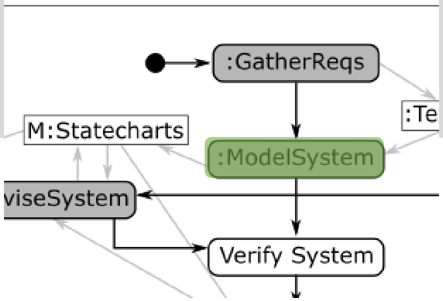


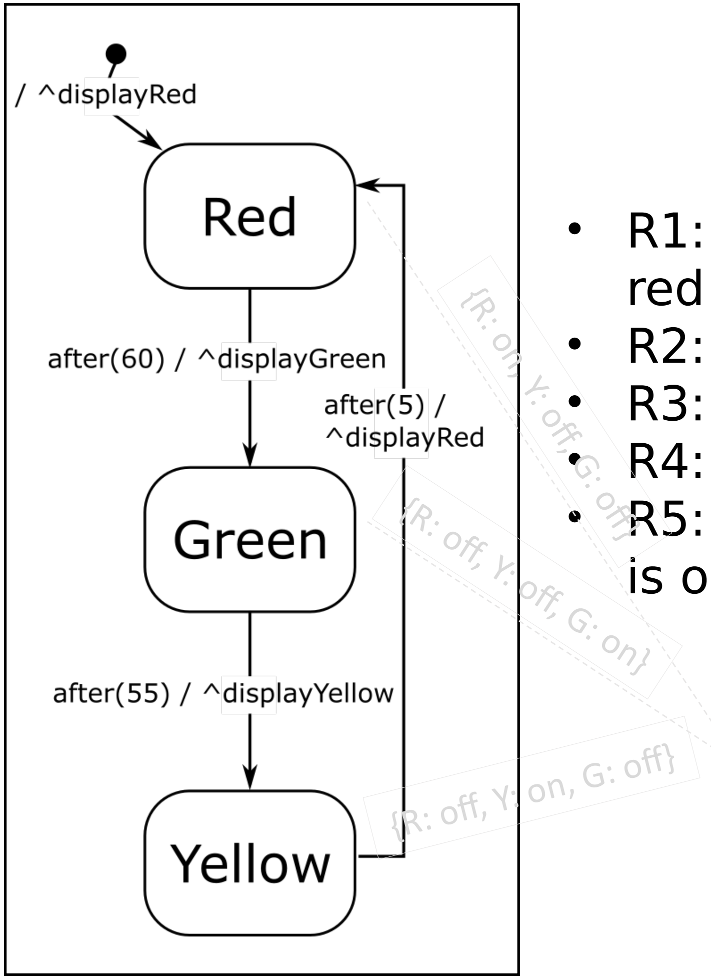
Your model here.

- R1: three differently coloured lights:
  red (R), green (G), yellow (Y)
- R2: at most one light is on at any point in time
- R3: at system start-up, the red light is on
- R4: cycles through red on, green on, and yellow on
- R5: red is on for 60s, green is on for 55s, yellow is on for 5s



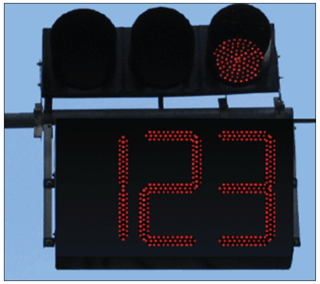
Exercise 1 - Solution





- R1: three differently coloured **lights**: red (R), green (G), yellow (Y)
- R2: at most one light is on at any point in time
- R3: at system start-up, the **red** light is on
- R4: cycles through red on, green on, and yellow on
- R5: **red** is on for 60s, **green** is on for 55s, **yellow** is on for 5s

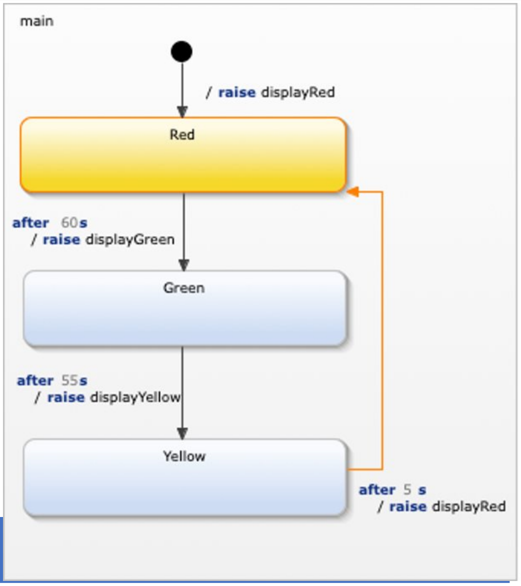
(Simulated) Plant



<<observe>>

Environment

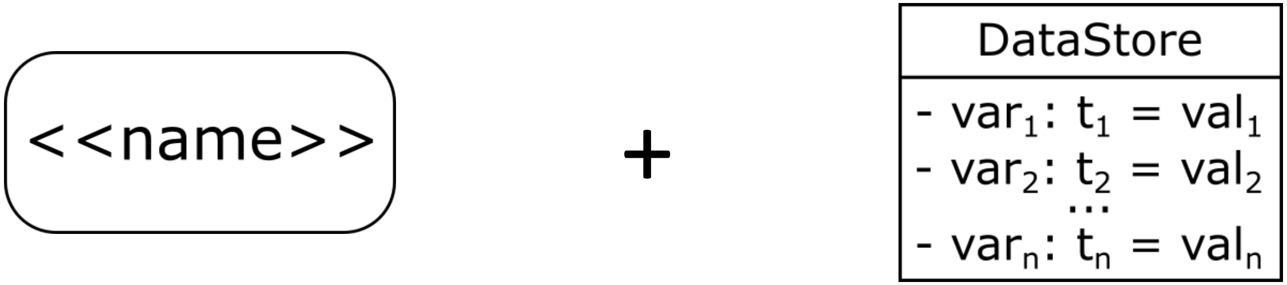
Exercise 1 - Solution



requirement	modelling approach
R1: three differently coloured **lights**: red (R), green (G), yellow (Y)	For each colour a **state** is defined. Transitions that lead to a state raise the proper out event which interacts with the plant.
R2: at most one light is on at any point in time	The states are all contained in a single region and thus a exclusive to each other ("or" states).
R3: at system start-up, the red light is on	The entry node points to state Red and the entry transition raises the event displayRed.
R4: cycles through red on, green on, and yellow on	The transitions define this cycle.
R5: red is on for 60s, green is on for 55s, yellow is on for 5s	Time events are specified on the transitions.

Data Store

Full System State



<<name>>

+

DataStore

- var_1: t_1 = val_1
- var_2: t_2 = val_2

...

- var_n: t_n = val_n

being **in** a state

= state <<*name*>> is **active**

= the system is in **configuration** <<*name*>>

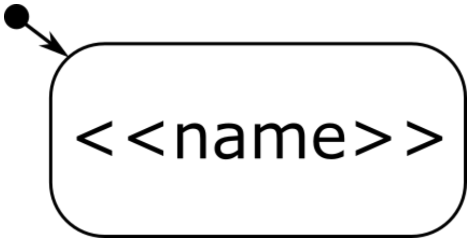
data store **snapshot**

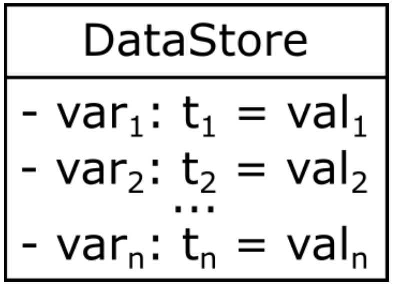
= variable values

=

full system state

Full System State: Initialization

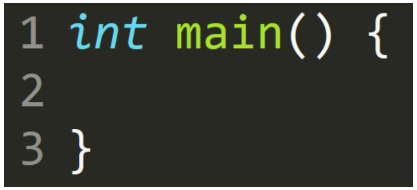



DataStore

- var_1: t_1 = val_1
- var_2: t_2 = val_2

...

- var_n: t_n = val_n



```
int main() {

}
```

initial state

exactly one per model

“entry point”

provide **default value** for each variable

“initial snapshot”

Compare:

C++ initialization

implicit state

(program counter)

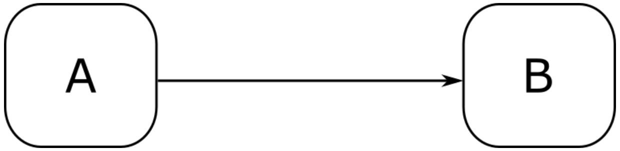
+ **data store**

Transitions: Guards

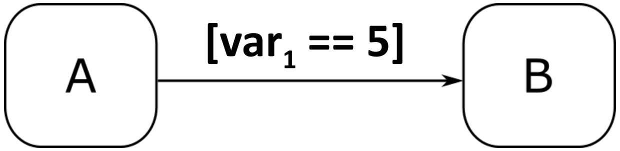
event(in_params) [guard] / output_action(out_params)

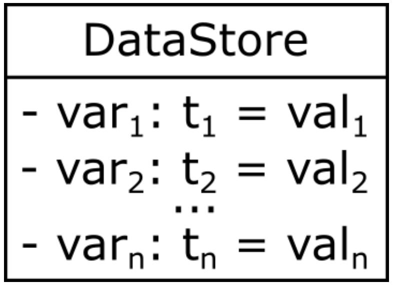
Modelled by “guard expression” (evaluates to Boolean) in some appropriate language

- Spontaneous [True]

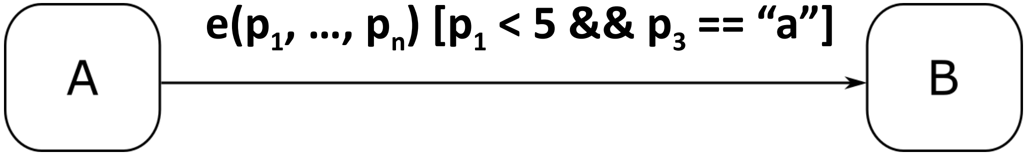


- Data Store Variable Value





- Parameter Value

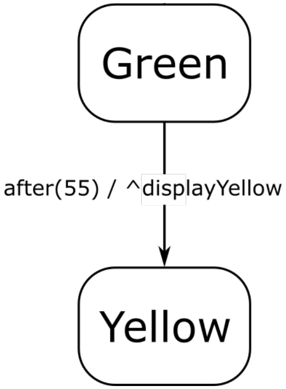


Transitions: Output Actions

event(params) [guard] / output_action(params)

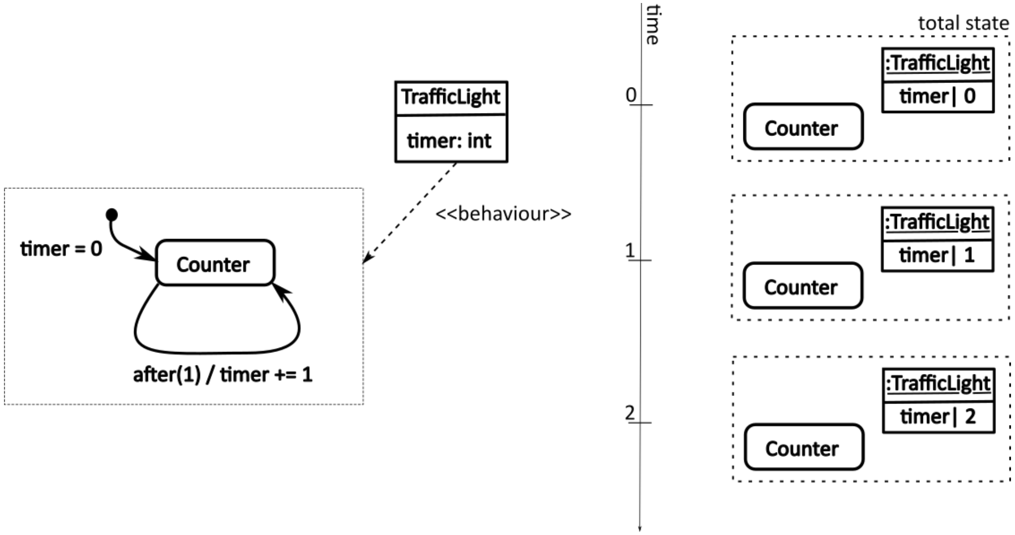
Output Event

^*output_event*(p_1, p_2, ..., p_n)

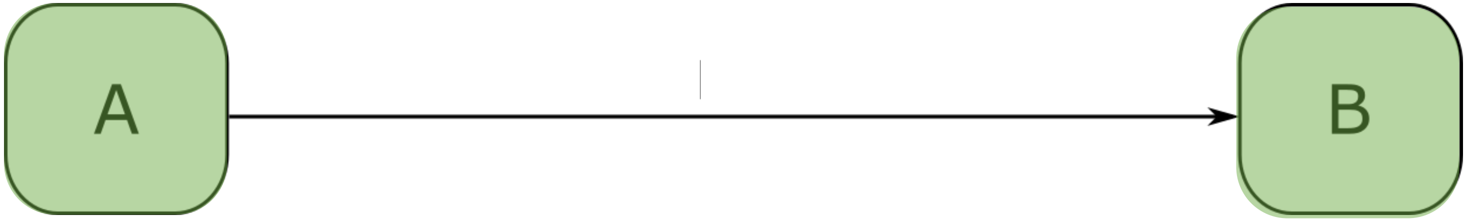


Assignment (to the non-modal part of the state)

- by action code in some appropriate language



Transitions

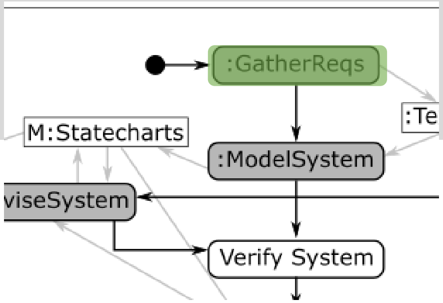


- Model the **dynamics** of the system:
  - *when*
    - the system is **in state A** and
    - **event is received** and
    - **guard** evaluates to **True**
  - *then*
    1. **output_action** is evaluated and
    2. the new **active state** becomes B

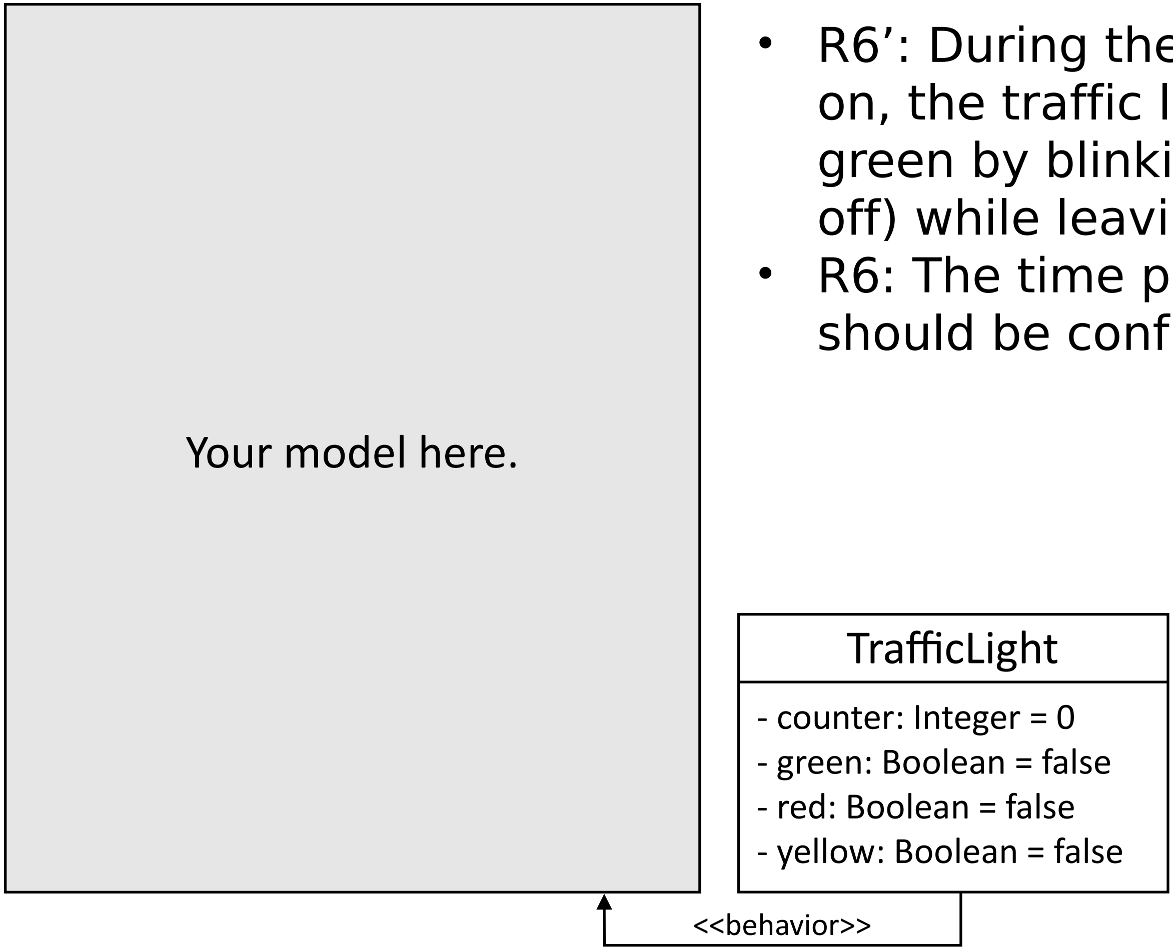
Exercise 2

Add data stores

Exercise 2 - Requirements



- R6’: During the last 6 seconds of red being on, the traffic light announces to go to green by blinking its yellow light (1s on, 1s off) while leaving its red light on.
- R6: The time period of the different phases should be configurable.



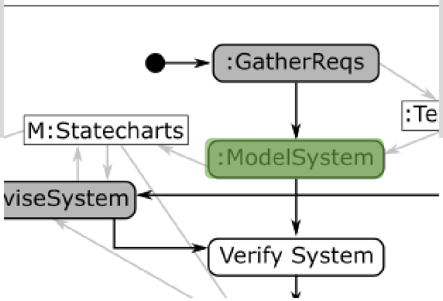
TrafficLight

- counter: Integer = 0
- green: Boolean = false
- red: Boolean = false
- yellow: Boolean = false

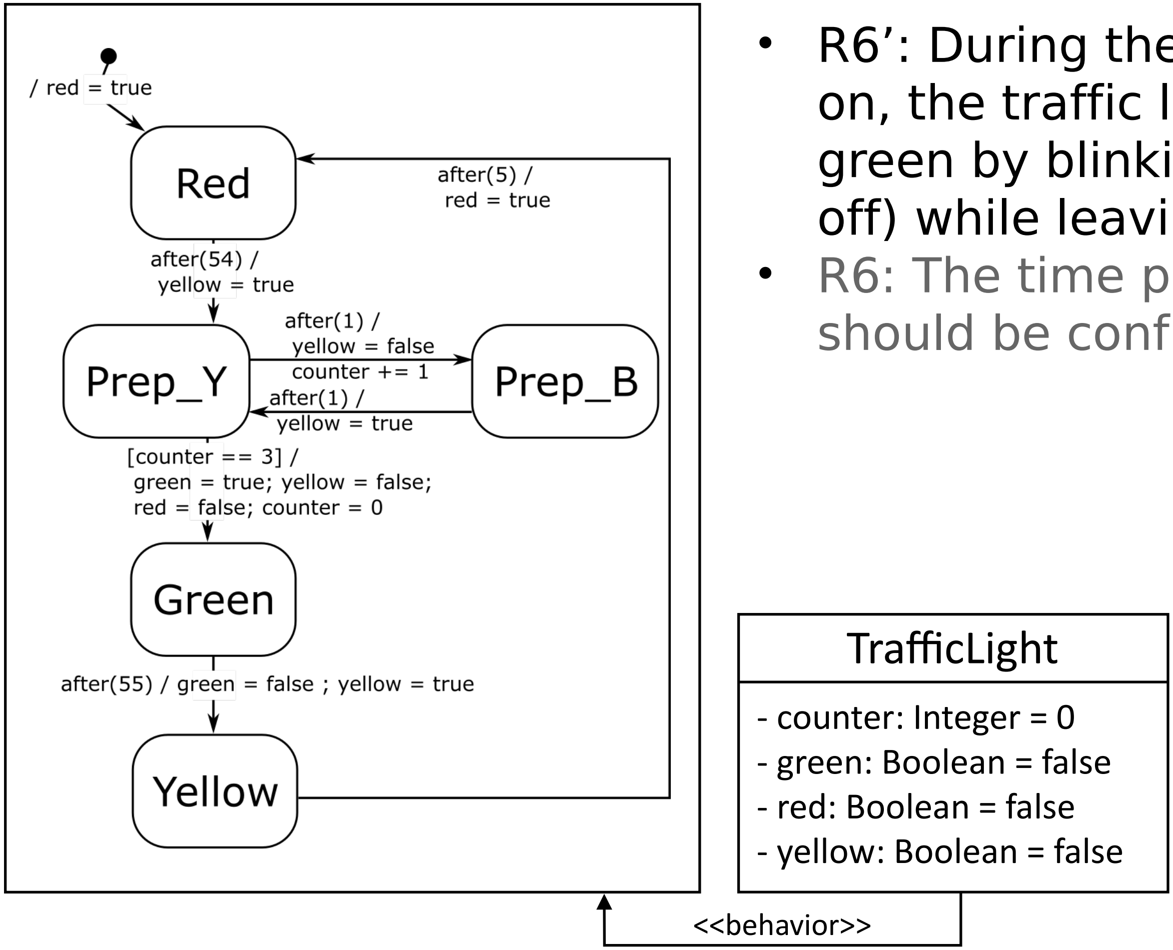
Make sure that:

- **the values of the variables reflect which lights are on/off**
- **you use at least one conditional transition**

Exercise 2: Solution



- R6’: During the last 6 seconds of red being on, the traffic light announces to go to green by blinking its yellow light (1s on, 1s off) while leaving its red light on.
- R6: The time period of the different phases should be configurable.



Statechart Execution

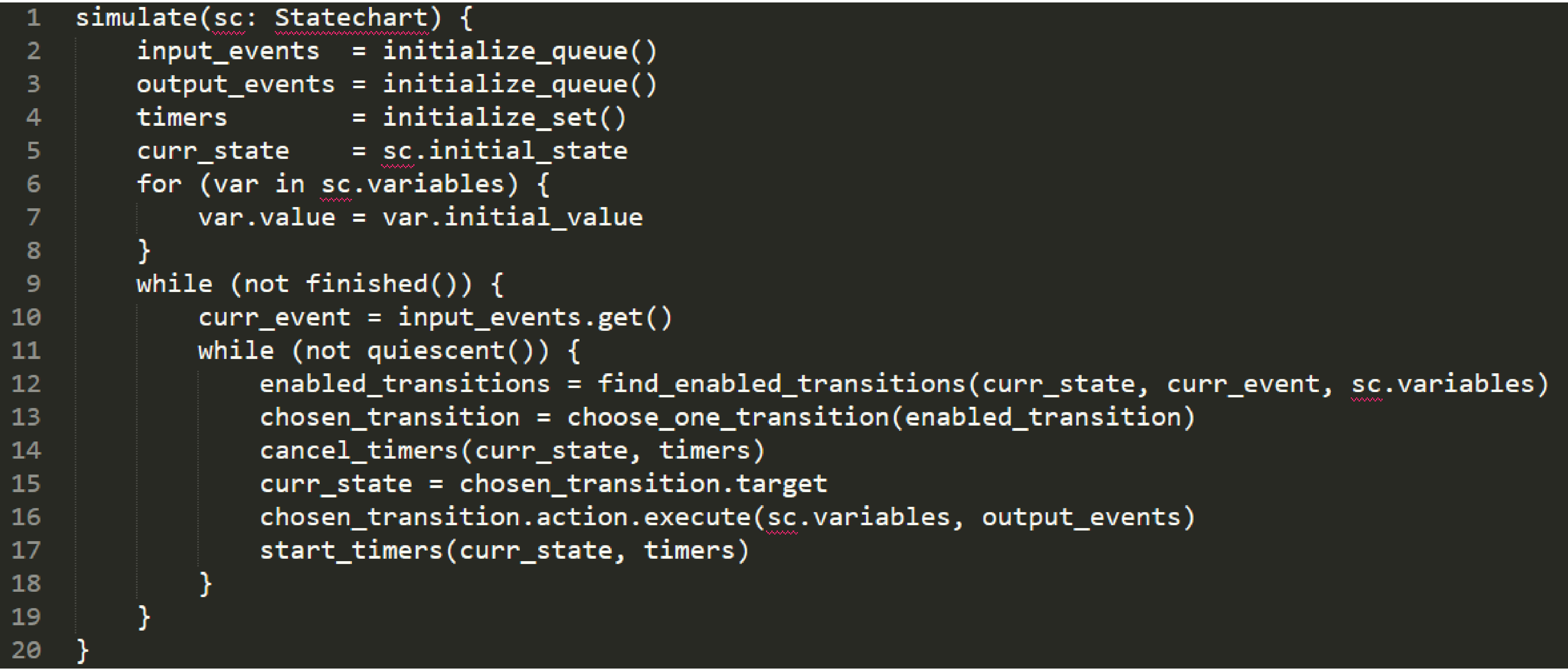
Run-To-Completion Step

- A Run-To-Completion (RTC) step is an *atomic execution step* of a state machine.
- It transitions the state machine from a *valid state configuration* into the next *valid state configuration*.
- RTC steps are executed one after the other - they must *not interleave*.
- New incoming events *cannot interrupt* the processing of the current event and must be stored in an *event queue*

Flat Statecharts: Simulation Algorithm (1)

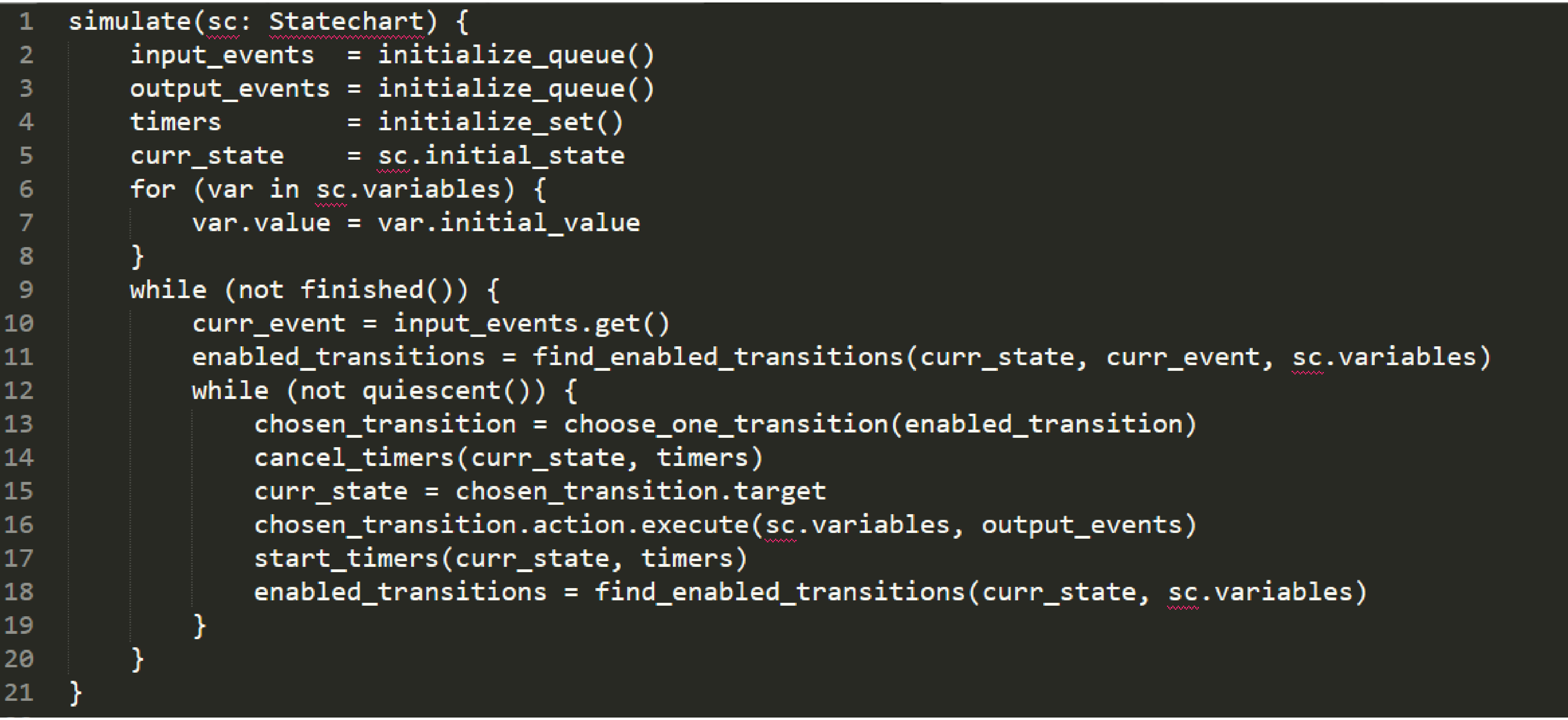
```
simulate(sc: Statechart) {

}
```

Flat Statecharts: Simulation Algorithm (2)



```
simulate(sc: Statechart) {
    input_events  = initialize_queue()
    output_events = initialize_queue()
    timers        = initialize_set()
    curr_state    = sc.initial_state
    for (var in sc.variables) {
        var.value = var.initial_value
    }
    while (not finished()) {
        curr_event = input_events.get()
        while (not quiescent()) {
            enabled_transitions = find_enabled_transitions(curr_state, curr_event, sc.variables)
            chosen_transition = choose_one_transition(enabled_transition)
            cancel_timers(curr_state, timers)
            curr_state = chosen_transition.target
            chosen_transition.action.execute(sc.variables, output_events)
            start_timers(curr_state, timers)
        }
    }
}
```

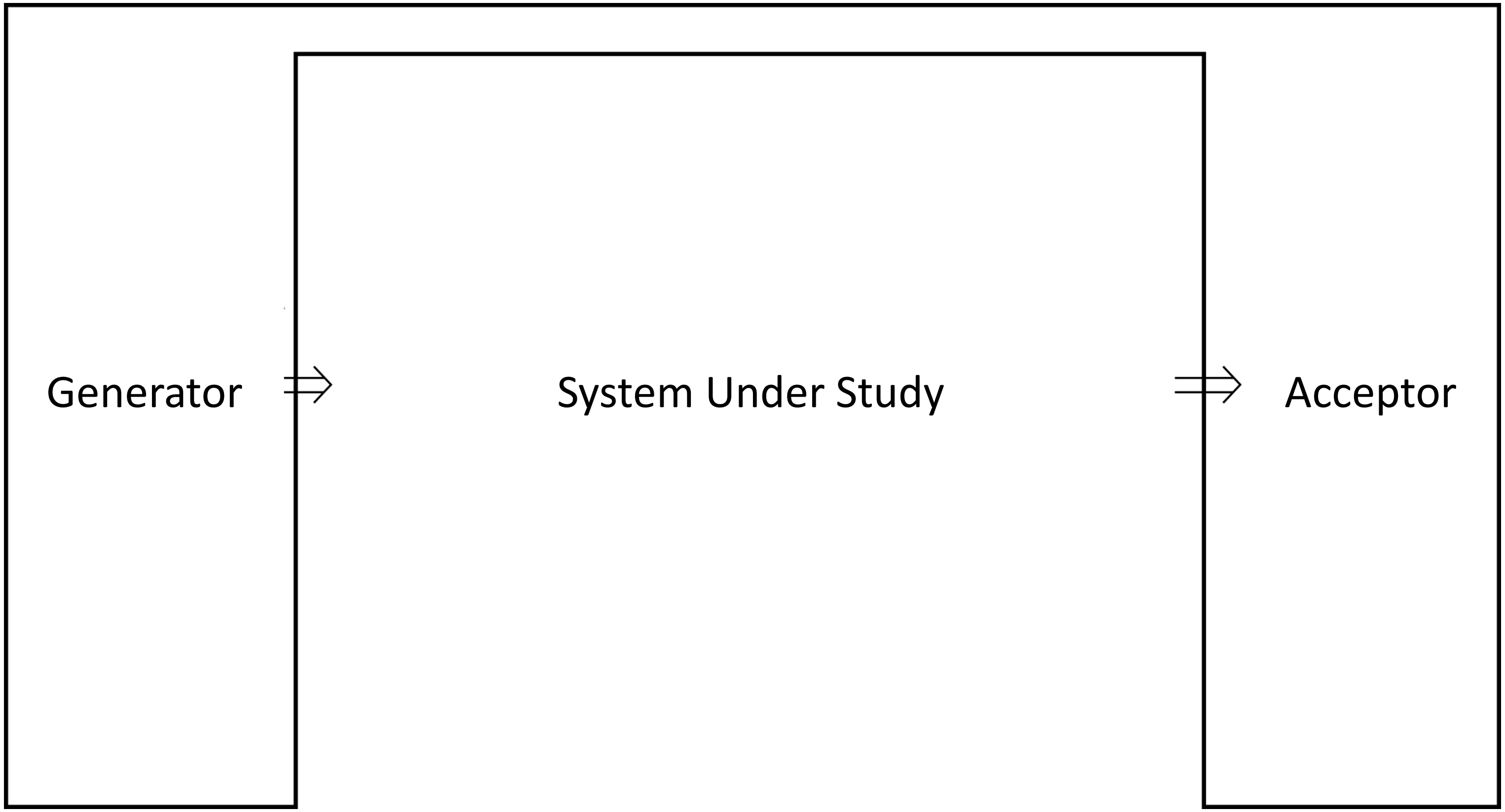
Flat Statecharts: Simulation Algorithm (3)



```
simulate(sc: Statechart) {
    input_events  = initialize_queue()
    output_events = initialize_queue()
    timers        = initialize_set()
    curr_state    = sc.initial_state
    for (var in sc.variables) {
        var.value = var.initial_value
    }
    while (not finished()) {
        curr_event = input_events.get()
        enabled_transitions = find_enabled_transitions(curr_state, curr_event, sc.variables)
        while (not quiescent()) {
            chosen_transition = choose_one_transition(enabled_transition)
            cancel_timers(curr_state, timers)
            curr_state = chosen_transition.target
            chosen_transition.action.execute(sc.variables, output_events)
            start_timers(curr_state, timers)
            enabled_transitions = find_enabled_transitions(curr_state, sc.variables)
        }
    }
}
```

Testing Statecharts

Testing Statecharts

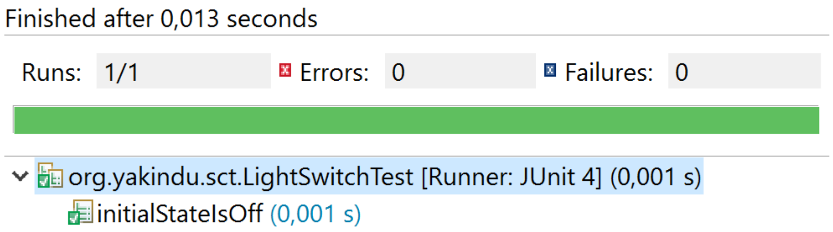


Zeigler BP. Theory of modelling and simulation. New York: Wiley-Interscience, 1976.

Mamadou K. Traoré, Alexandre Muzy, Capturing the dual relationship between simulation models and their context, Simulation Modelling Practice and Theory, Volume 14, Issue 2, February 2006, Pages 126-142.

SCTUnit (beta)

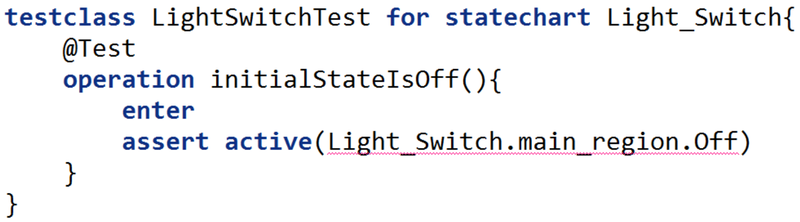
- X-unit testing framework for YAKINDU Statechart Tools
- Test-driven development of Statechart models
- Test generation for various platforms
- Executable in YAKINDU Statechart Tools
- Virtual Time



Finished after 0,013 seconds

Runs: 1/1 Errors: 0 Failures: 0

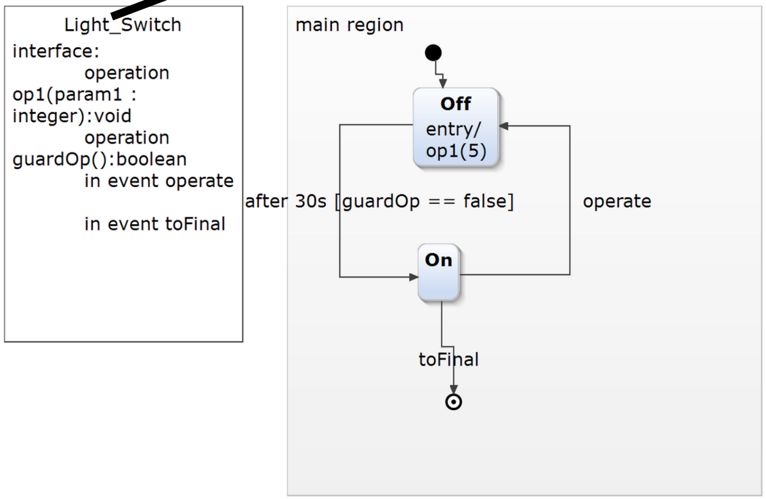
org.yakindu.sct.LightSwitchTest [Runner: JUnit 4] (0,001 s)
initialStateIsOff (0,001 s)



```
testclass LightSwitchTest for statechart Light_Switch{
    @Test
    operation initialStateIsOff(){
        enter
        assert active(Light_Switch.main_region.Off)
    }
}
```

Testclass

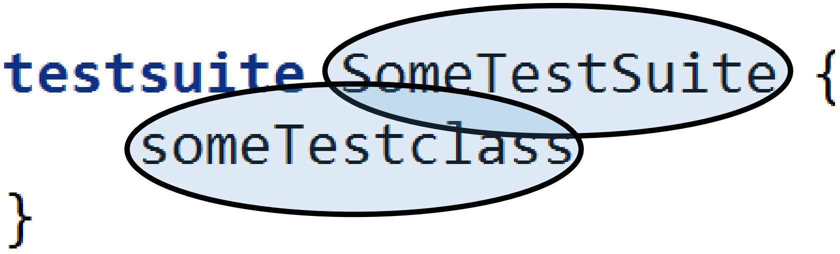
```
testclass someTestclass for statechart Light_Switch {

}
```



```
Light_Switch
interface:
        operation
op1(param1 :
integer):void
        operation
guardOp():boolean
        in event operate

        in event toFinal
```

- Has a unique name
- Has a reference to a statechart
- Contains one or more operations

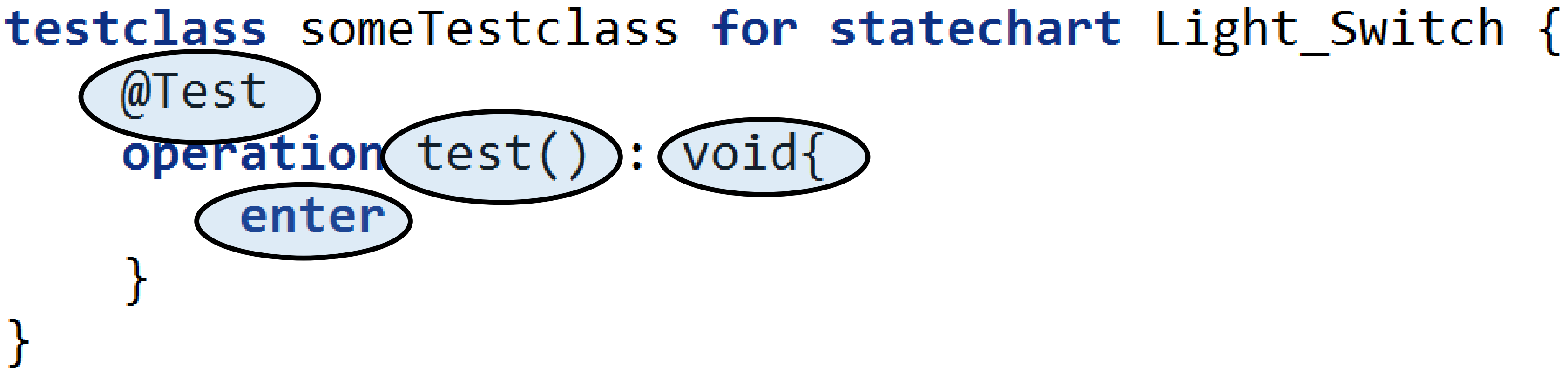
Testsuite



```
testsuite SomeTestSuite {
    someTestclass
}
```

- Has a unique name
- A testsuite contains at least one reference to a testclass

Operation



```
testclass someTestclass for statechart Light_Switch {
    @Test
    operation test() : void{
        enter
    }
}
```

- May have @Test or @Run annotation
- Has a unique name
- May have 0..n parameters
- Has a return type (standard is void)
- Contains 0..n statements

Expressions

```
// entering / exiting the statechart
enter, exit
// raising an event
raise event : value

// proceeding time or cycles
proceed 2 cycle
proceed 200 ms

// asserting an expression, expression must evaluate to boolean
assert expression
// is a state active
active(someStatechart.someRegion.someState)
```

Mocking Statements

SCTUnit allows to

- mock operations defined in the statechart model
- verify that an operation was called with certain values

```
// mocking the return value of an operation
mock mockOperation returns (20)
mock mockOperation(5) returns (30)

// verifying the call of an operation
assert called verifyOperation
assert called verifyOperation with (5, 10)
```

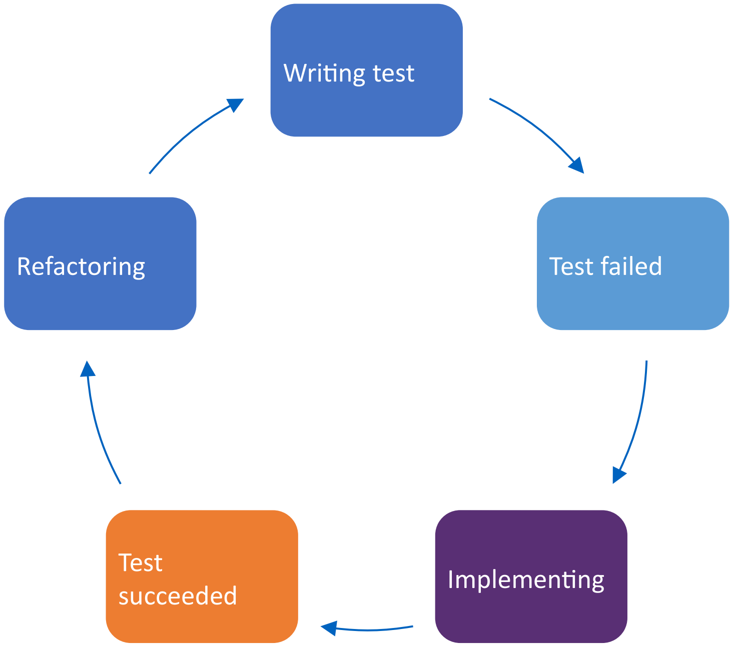
Control Structures

```
// if expression

if (x==5) {
    doSomething()
} else {
    doSomethingelse()
}
```

```
// while expression

while (x==5) {
    doSomething()
}
```

Test-Driven Development

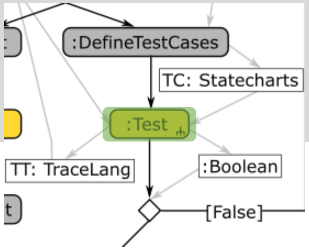
- Software development process, where software is developed driven by tests
- Test-first-approach
- 3 steps you do repeatedly:
    - writing a test
    - implementing the logic
    - refactoring

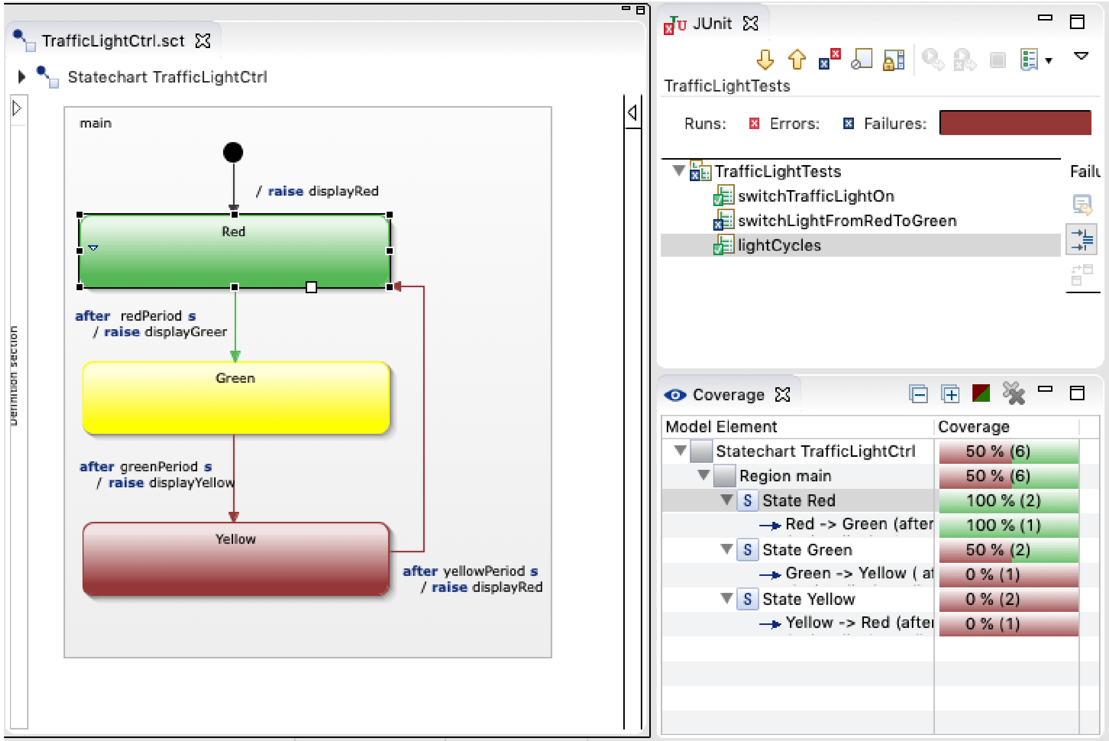


Exercise 3

Testing Models

Exercise 3 – Unit testing Statecharts

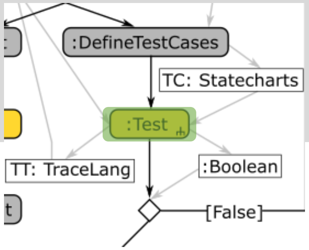




Model Element	Coverage
Statechart TrafficLightCtrl	50 % (6)
Region main	50 % (6)
S State Red	100 % (2)
Red -> Green (after	100 % (1)
S State Green	50 % (2)
Green -> Yellow (af	0 % (1)
S State Yellow	0 % (2)
Yellow -> Red (after	0 % (1)

- Create a test that checks the following requirements:
  - R3: at system start-up, the red light is on
  - R4: cycles through red on, green on, and yellow on
  - R5: red is on for 60s, green is on for 55s, yellow is on for 5s

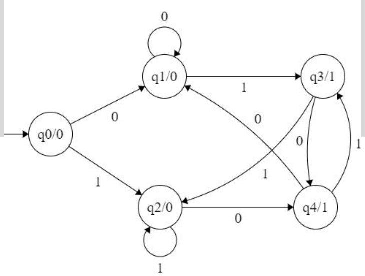
Exercise 3 – Solution



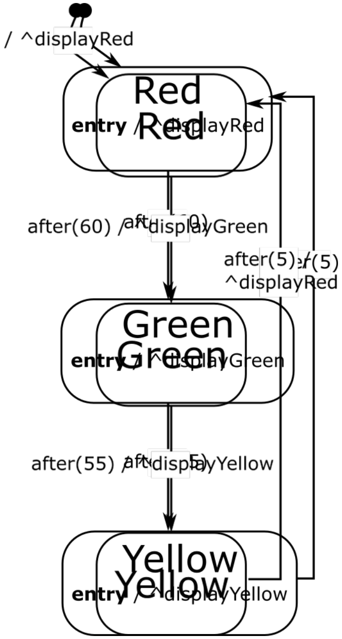
```
package trafficlight.test

testclass TrafficLightTests for statechart TrafficLightCtrl {

    @Test operation switchTrafficLightOn () {

        // given the traffic light is inactive
        assert !is_active
        // when
        enter
        // then traffic light is off which means no color was switched on
        assert displayRed
        assert !displayGreen
        assert !displayYellow
    }


    @Test operation switchLightFromRedToGreen () {

        // given
        switchTrafficLightOn
        // when
        proceed 60s
        // then
        assert displayGreen
    }


    @Test operation switchLightFromGreenToYellow () {

        // given
        switchLightFromRedToGreen
        // when
        proceed 55s
        // then
        assert displayYellow
    }


    @Test operation switchLightFromYellowToRed () {

        // given
        switchLightFromGreenToYellow
        // when
        proceed 5s
        // then
        assert displayRed
    }

    @Test operation lightCycles () {

        // given
        switchLightFromYellowToRed

        var i : integer = 10

        while (i > 0) {
            i=i-1

            //when
            proceed 60 s
            // then
            assert displayGreen

            //when
            proceed 55 s
            // then
            assert displayYellow

            //when
            proceed 5 s
            // then
            assert displayRed
        }
    }
}
```

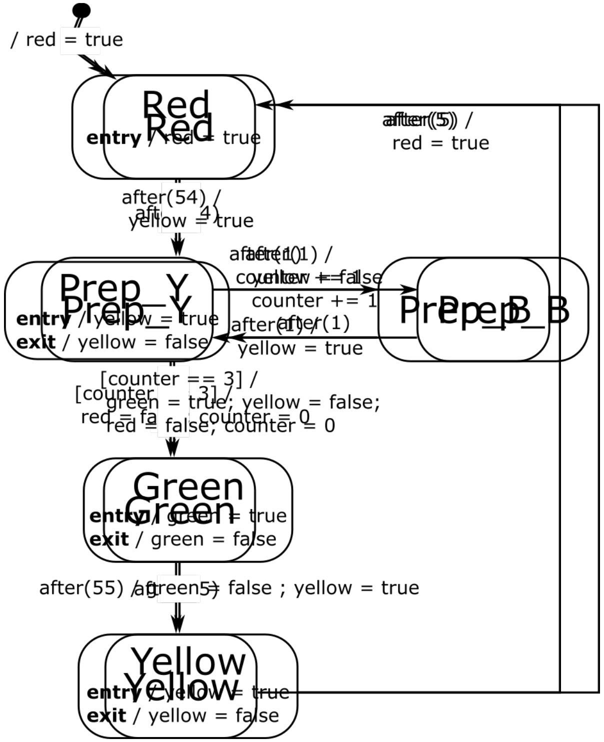
Hierarchy

Entry/Exit Actions

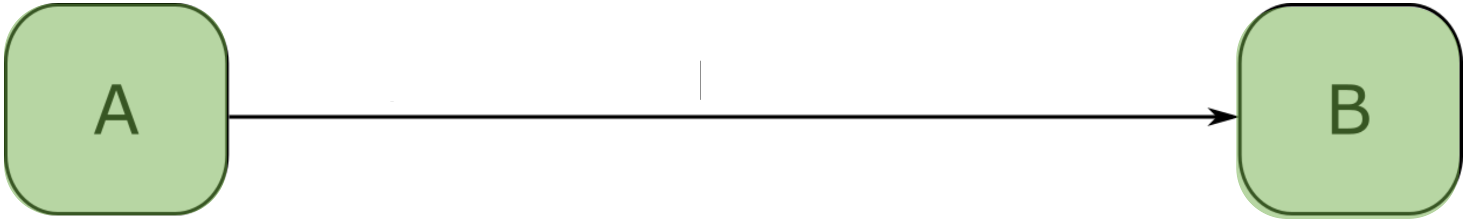


- A state can have entry and exit actions.
- An *entry action* is executed whenever a state is entered (made active).
- An *exit action* is executed whenever a state is exited (made *inactive*).
- Same expressiveness as *transition actions* (*i.e.,* syntactic sugar).



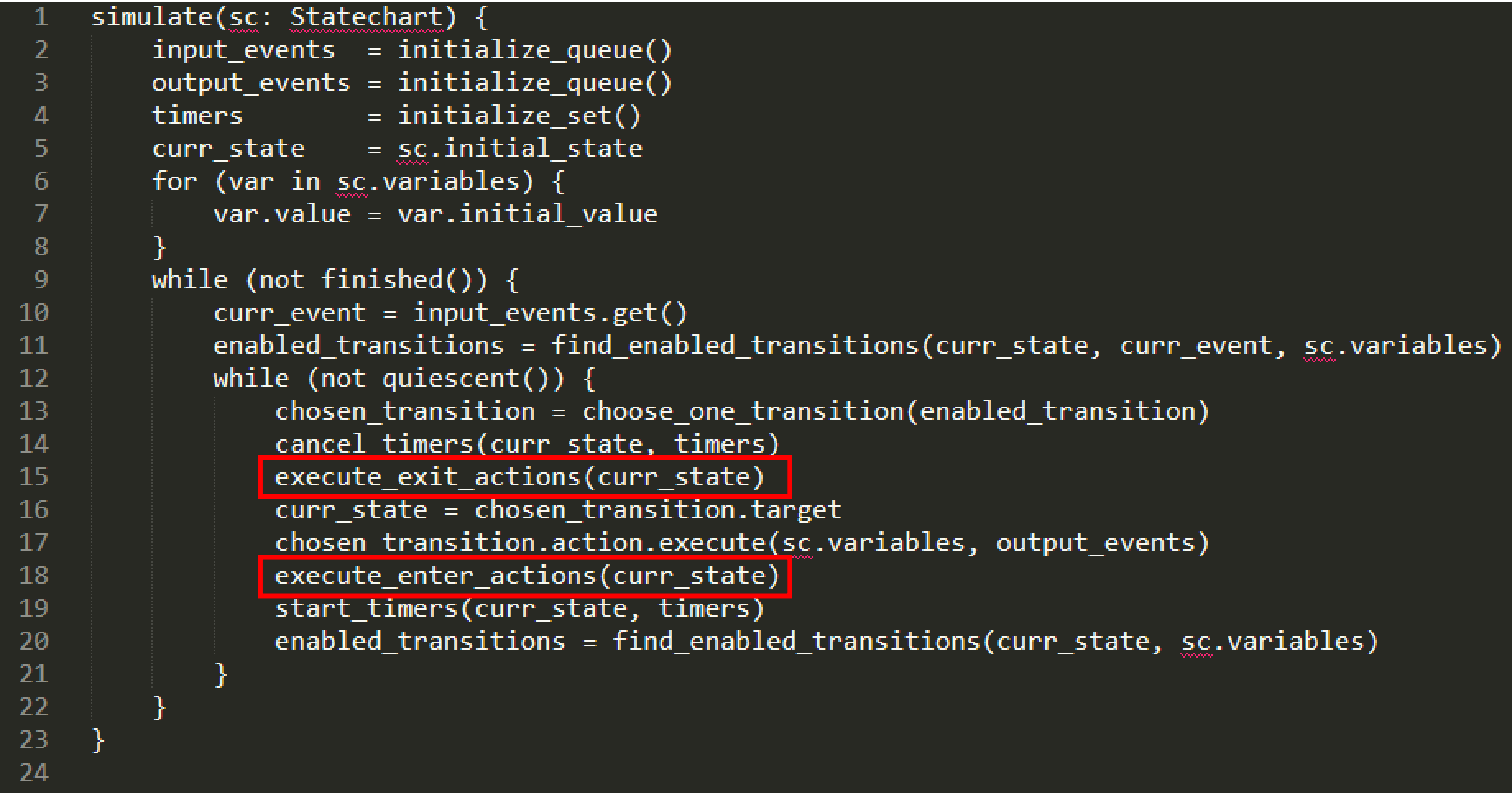


Transitions



- Model the **dynamics** of the system:
  - *when*
    - the system is **in state A** and
    - **event is received** and
    - **guard** evaluates to **true**
  - *then*
    1. the **exit actions** of state A are evaluated and
    2. **output_action** is evaluated and
    3. the **enter actions** of state B are evaluated and
    4. the new **active state** becomes B

Entry/Exit Actions: Simulation Algorithm

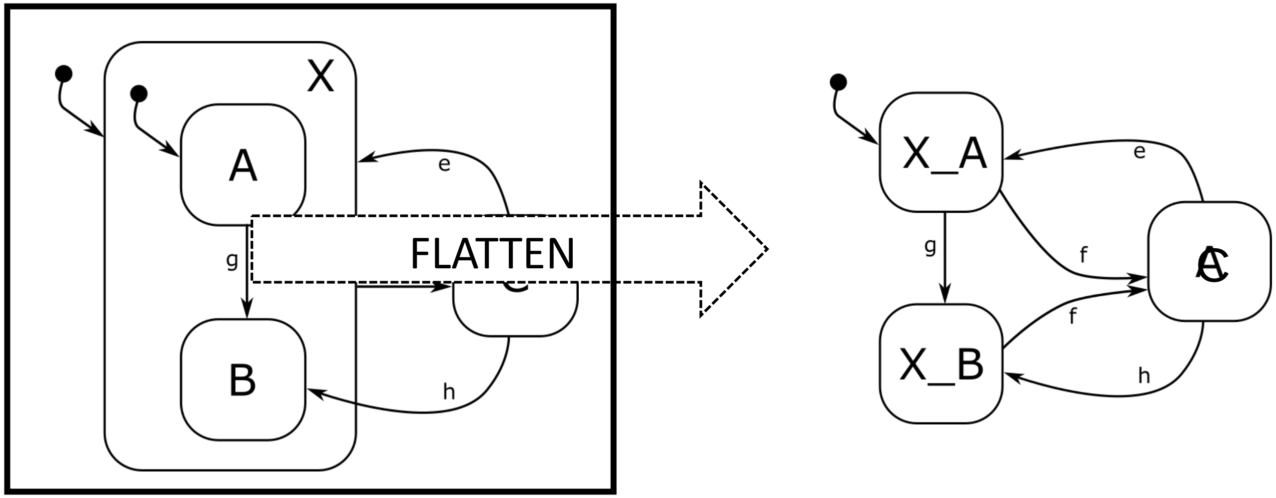


```
simulate(sc: Statechart) {
    input_events  = initialize_queue()
    output_events = initialize_queue()
    timers        = initialize_set()
    curr_state    = sc.initial_state
    for (var in sc.variables) {
        var.value = var.initial_value
    }
    while (not finished()) {
        curr_event = input_events.get()
        enabled_transitions = find_enabled_transitions(curr_state, curr_event, sc.variables)
        while (not quiescent()) {
            chosen_transition = choose_one_transition(enabled_transition)
            cancel_timers(curr_state, timers)
            execute_exit_actions(curr_state)
            curr_state = chosen_transition.target
            chosen_transition.action.execute(sc.variables, output_events)
            execute_enter_actions(curr_state)
            start_timers(curr_state, timers)
            enabled_transitions = find_enabled_transitions(curr_state, sc.variables)
        }
    }
}
```

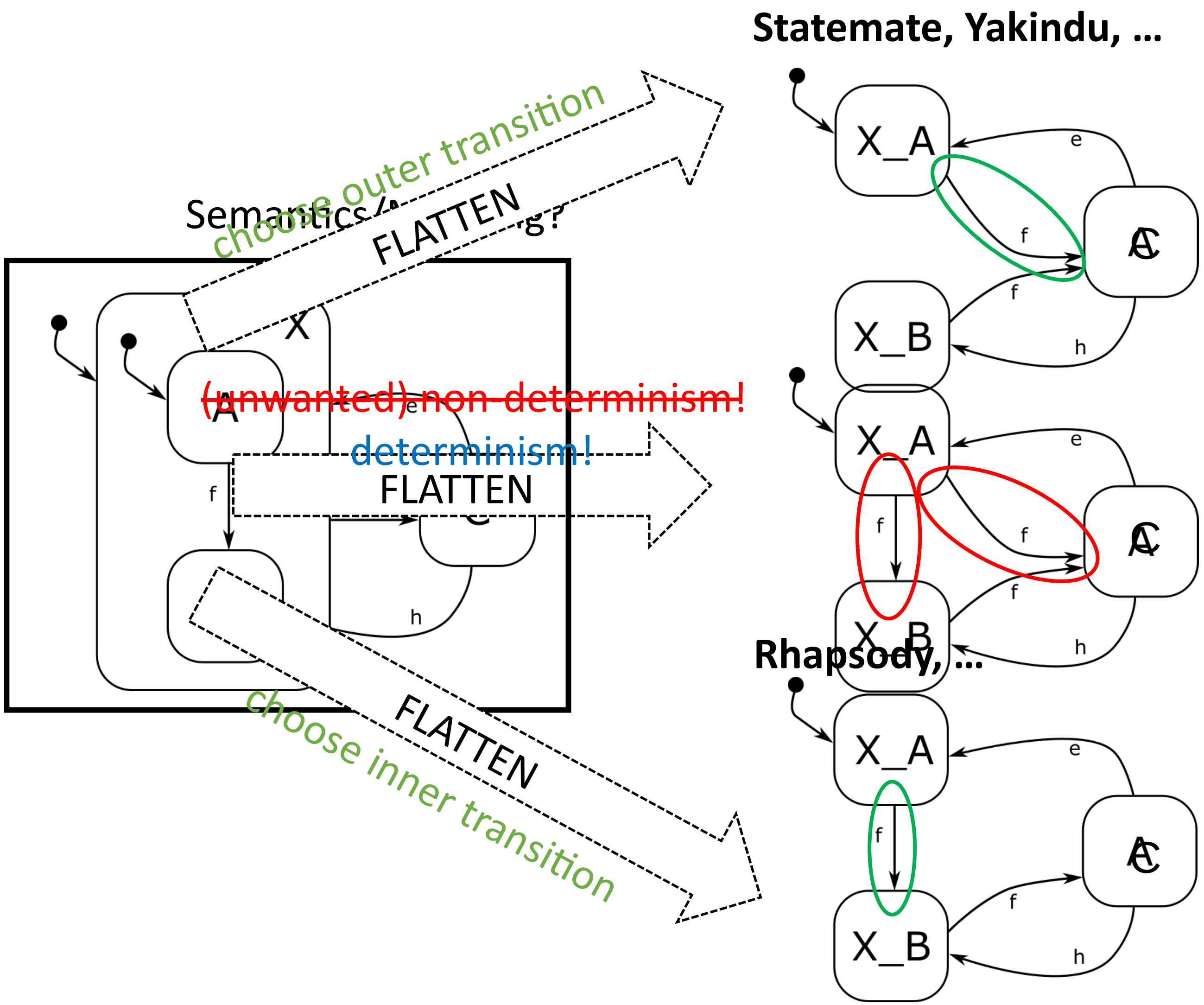
Hierarchy

- Statechart states can be hierarchically (de-)**composed**
- Each hierarchical state has exactly one **initial/default state**
- An active hierarchical state has **exactly one active child** (down to leaf/atomic state)

Semantics/Meaning?



Hiearchy: Modified Example



Statemate, Yakindu, …

Semantics/Meaning?

choose outer transition

FLATTEN

~~(unwanted) non-determinism!~~

determinism!

FLATTEN

FLATTEN

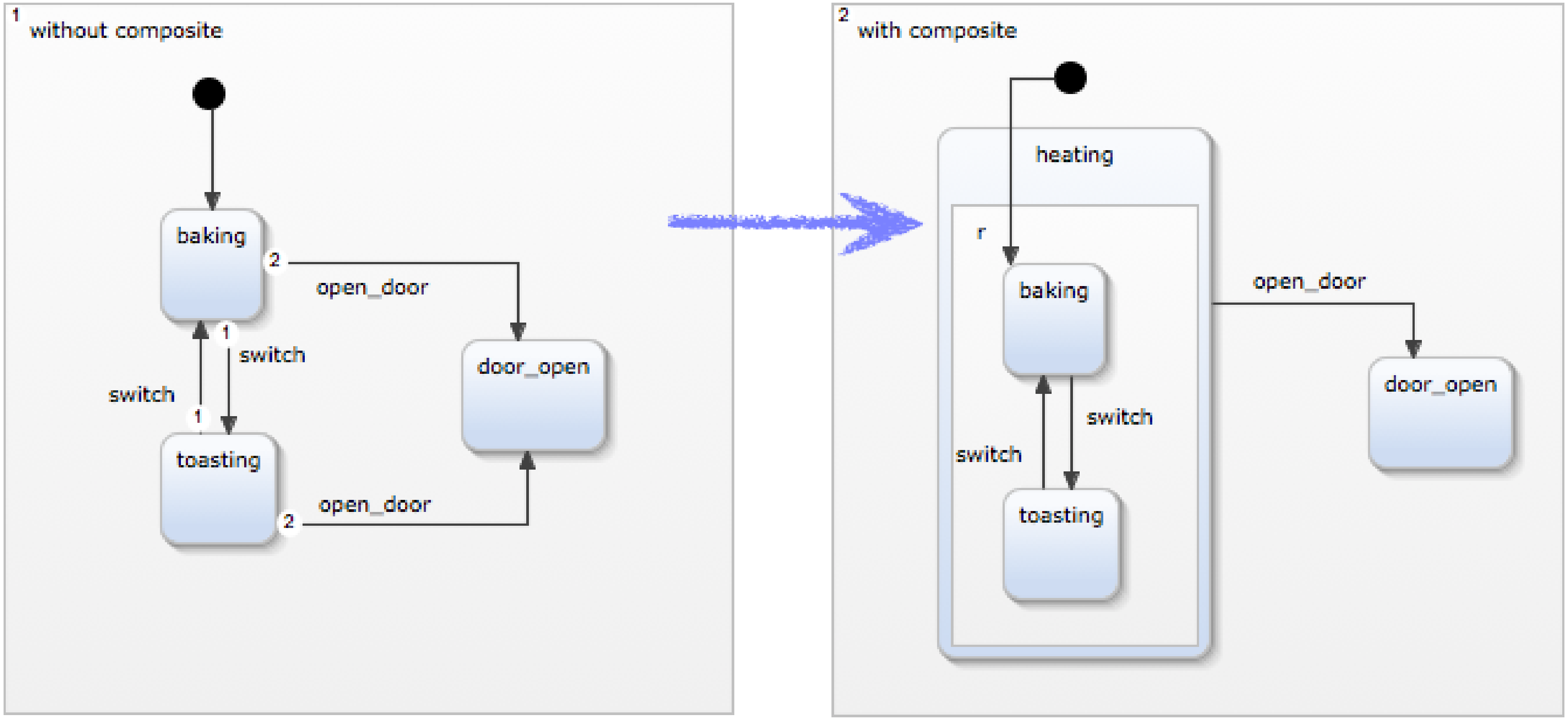
choose inner transition

Rhapsody, …

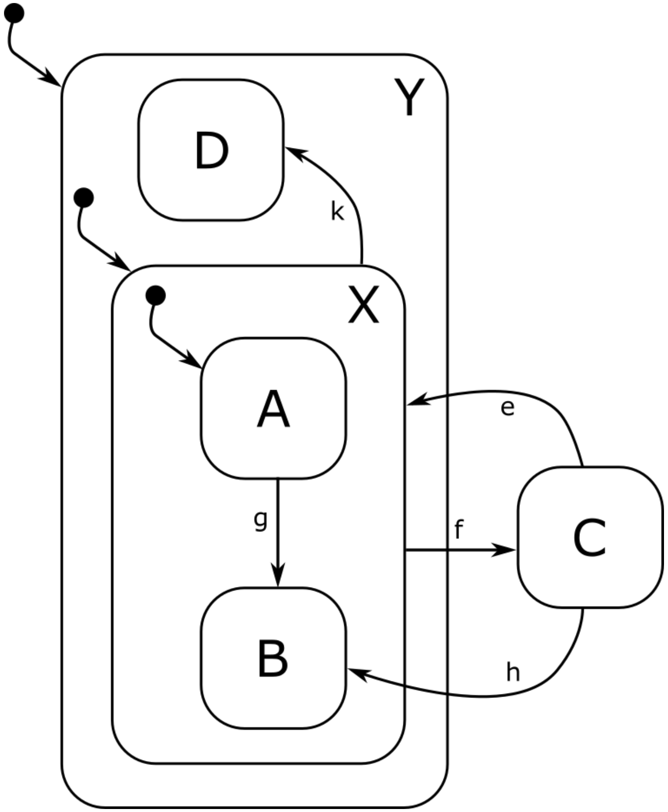
Hiearchy: why inner? ... see Code Generation

Composite States

- Hierarchical states are an ideal mechanism for hiding complexity
- Parent states can implement common behaviour for their substates
- Hierachical event processing **reduces the number of transitions**
- Refactoring support: group states into a composite state

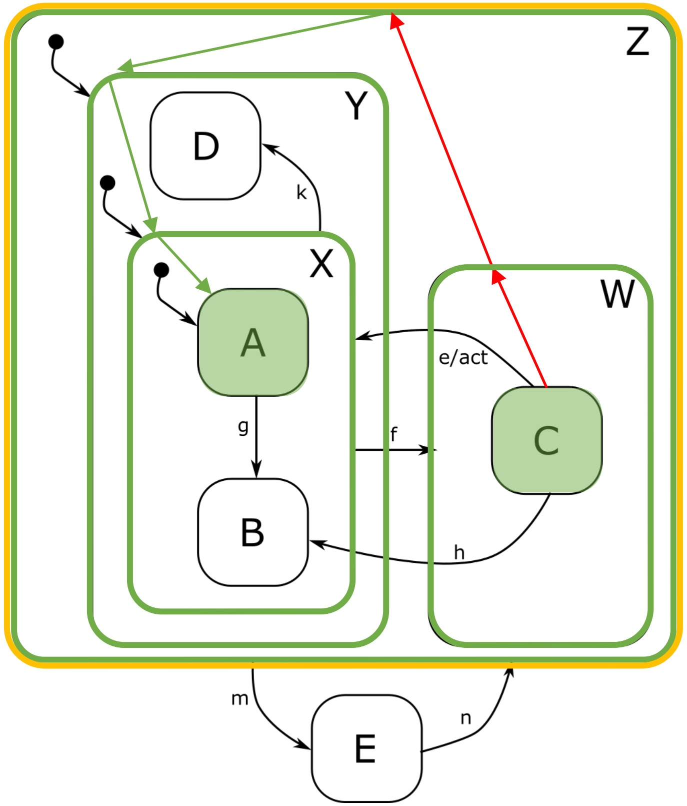


Hierarchy: Initialization



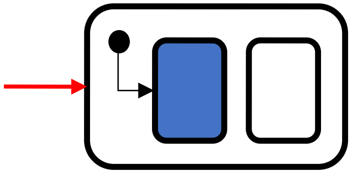
- Concept of **effective target state**
  - Recursive: the effective target state of a composite state is its initial state
- Effective target state of initial transition is *Y/X/A*
- Initialization:
  1. Enter Y, execute enter action
  2. Enter X, execute enter action
  3. Enter A, execute enter action

Hierarchy: Transitions



- Assume *Z/W/C* is active and *e* is processed.
- Semantics:
  1. Find LCA, collect states to leave
  2. Leave states up the hierarchy
  3. Execute action *act*
  4. Find effective target state set, enter states down the hierarchy

Effective target states:

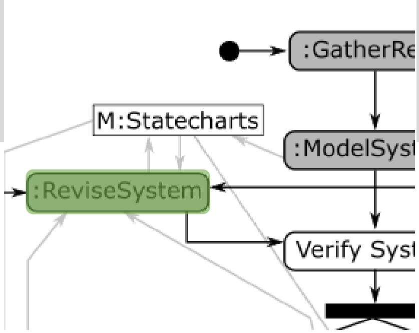


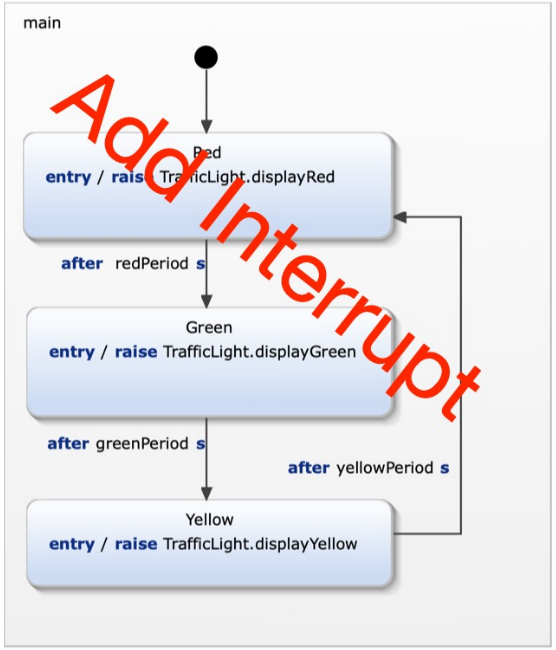
RECURSIVE!

Exercise 5

Model an interruptible traffic light

Exercise 5 - Requirements

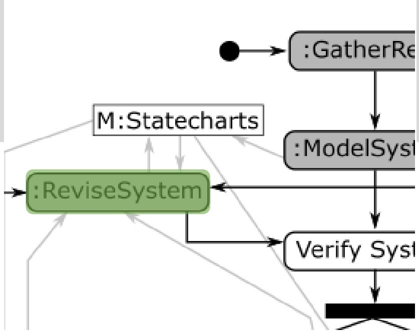




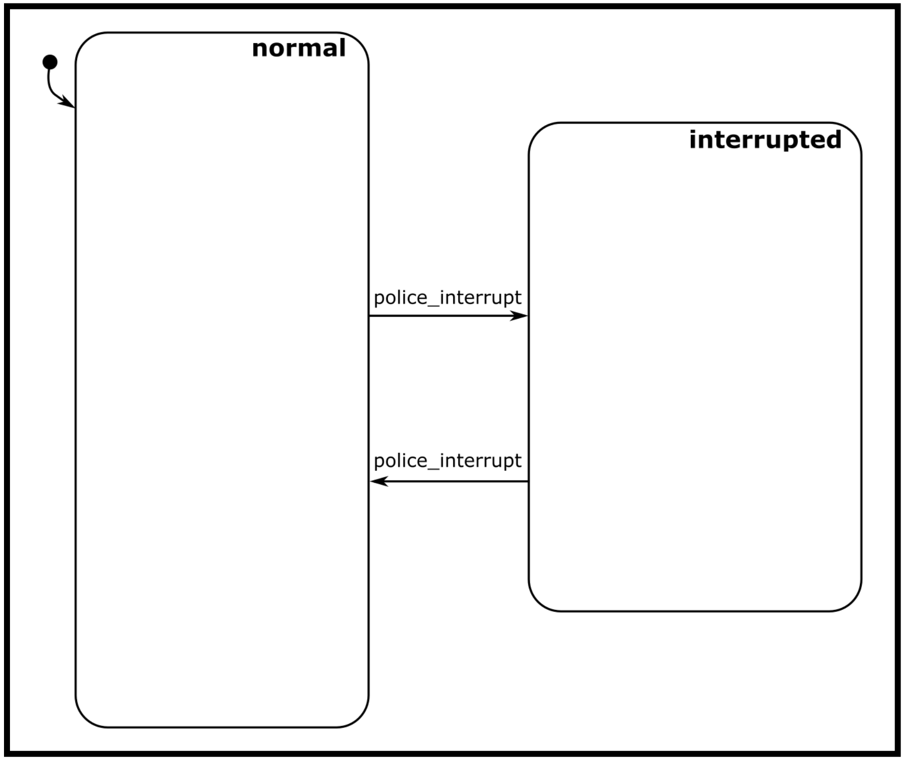
main
Add Interrupt
Red
entry / raise TrafficLight.displayRed
after redPeriod s
Green
entry / raise TrafficLight.displayGreen
after greenPeriod s
after yellowPeriod s
Yellow
entry / raise TrafficLight.displayYellow

- R7a: police can interrupt autonomous operation .
- R7b: autonomous operation can be interrupted during any phase of constant red, yellow and green lights.
- R7c: in interrupted mode the yellow light blinks with a constant frequency of 1 Hz (on 0.5s, off 0.5s).
- R8a: police can resume to regular autonomous operation.
- R8b: when regular operation is resumed, the traffic light restarts with red (R) light on.

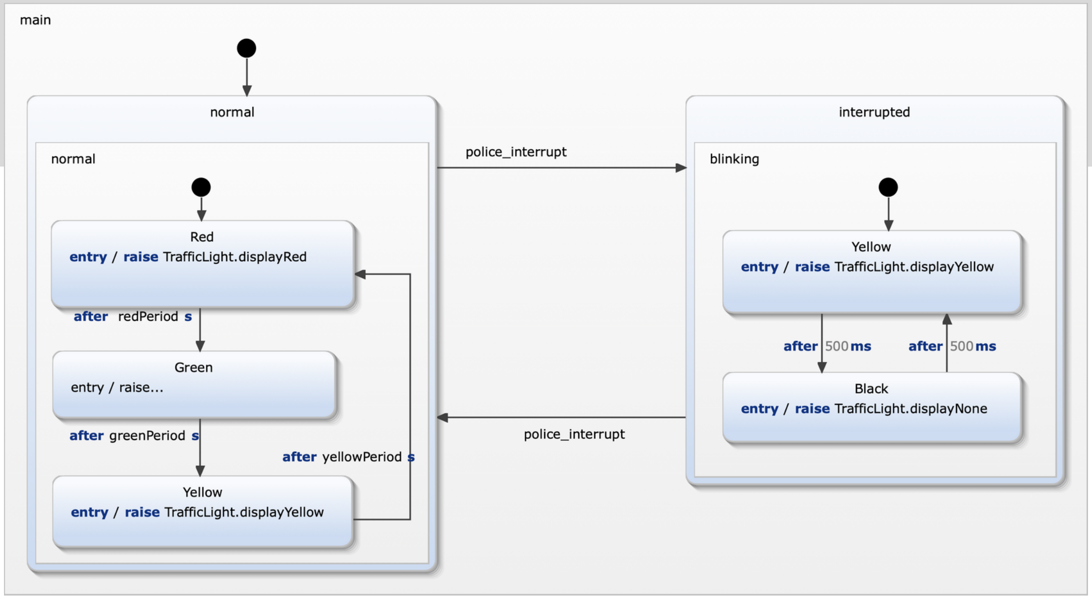
Exercise 5: Solution



- R7a: police can interrupt autonomous operation .
- R7b: autonomous operation can be interrupted during any phase of constant red, yellow and green lights.
- R7c: in interrupted mode the yellow light blinks with a constant frequency of 1 Hz (on 0.5s, off 0.5s).
- R8a: police can resume to regular autonomous operation.
- R8b: when regular operation is resumed, the traffic light restarts with red (R) light on.



Exercise 5 - Solution

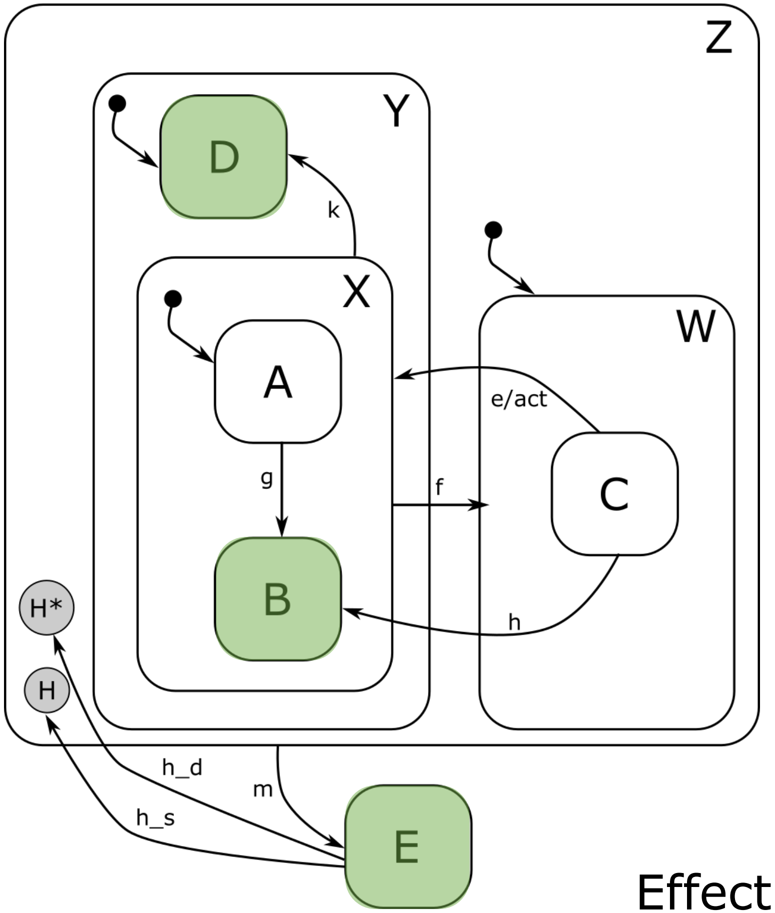


requirement	modelling approach
R6: police can interrupt autonomous operation.	An new incoming event police_interrupt triggers a transition to a new state interrupted.
R6a: autonomous operation can be interrupted during any phase of constant red, yellow and green lights.	The states Red, Green, and Yellow are grouped within a new composite state normal. This state is the source state of the transition to state interrupted and thus also applies to all substates.
R7: in interruptetd mode the yellow light blinks with a constant frequency of 1 Hz. (on 0.5s, off 0.5s).	State interrupted is a composite state with two substates Yellow and Black. These switch the yellow light on and off. Timed transitions between these states ensure correct timing for blinking.
R8: police can resume to regular autonomous operation.	A transition triggered by police_interrupt leads from state interrupted to state normal.
R8a: when regular operation is resumed the traffic light restarts with red (R) light on.	When activating state normal its substate Red is activated by default.

History

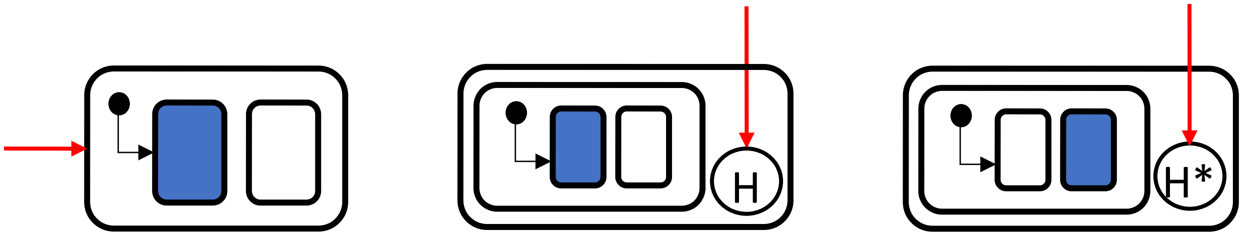
History: pseudo-states

H *shallow* history H* *deep* history



- Assume *Z/Y/X/B* is active, and *m* is processed
  - Effective target state: *E*
- If *h_s* is processed
  - Effective target state: *Z/Y/D*
- If *h_d* is processed
  - Effective target state: *Z/Y/X/B*

Effective target states:

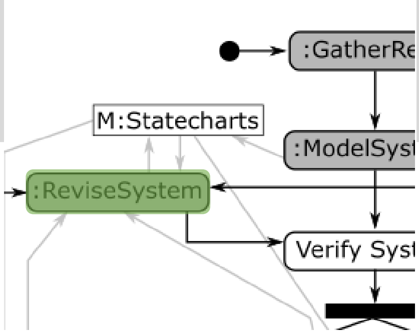


RECURSIVE!

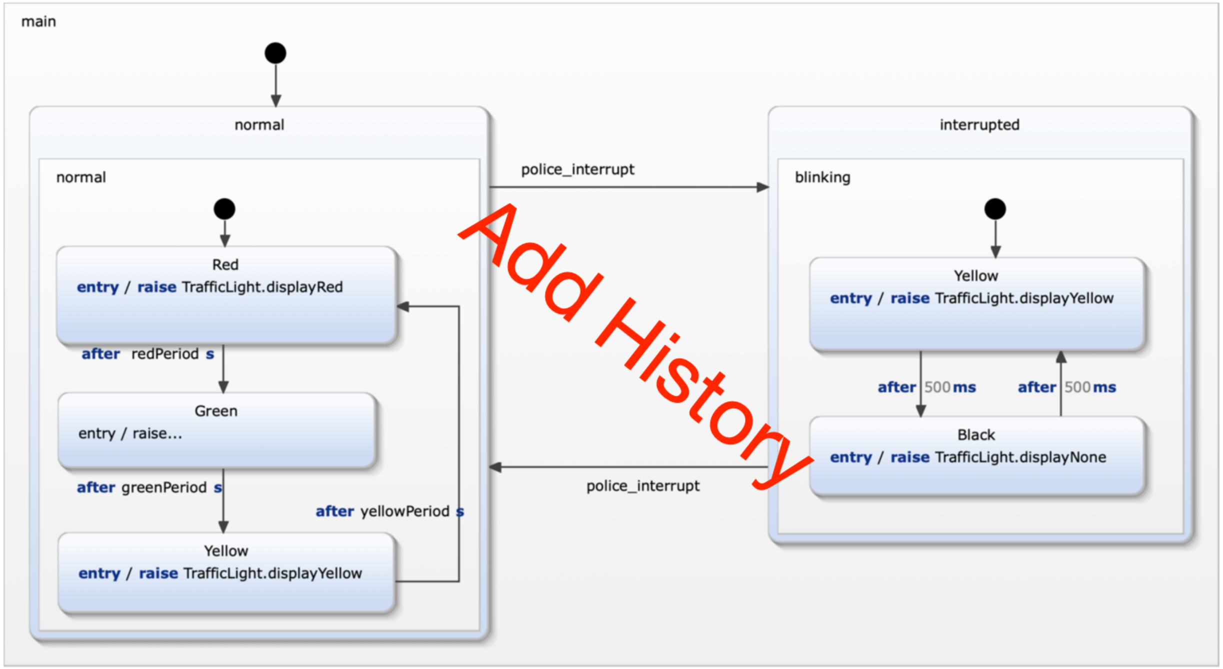
Exercise 6

Model an interruptible traffic light that restores its state

Exercise 6: Requirements

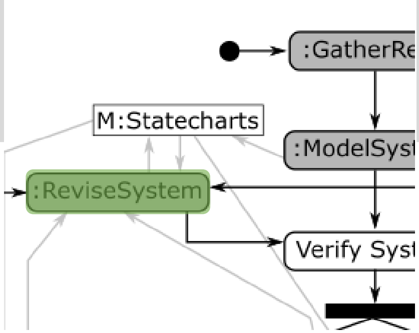


- R8b: when regular operation is resumed the traffic light restarts with the last active light color red (R), green (G), or yellow (Y) on.

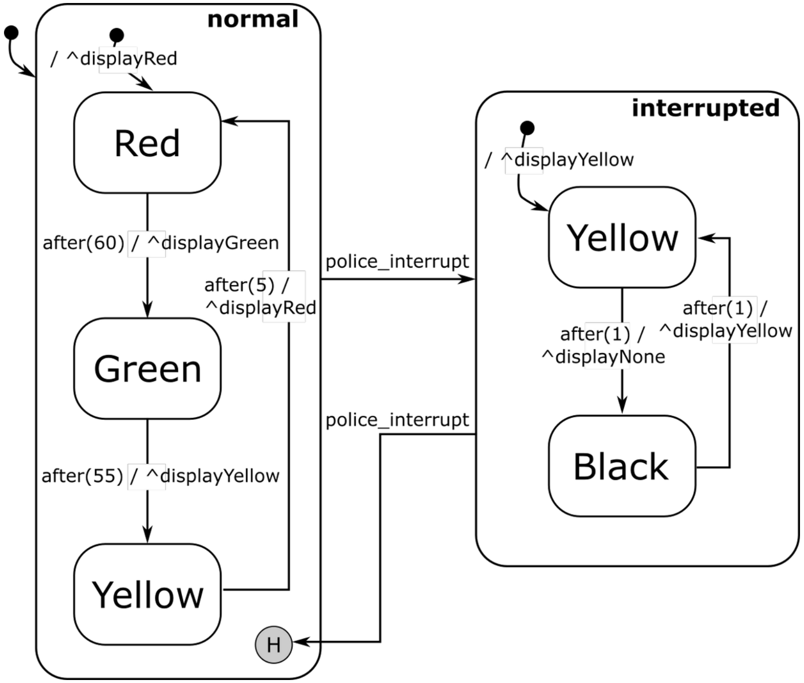


Add History

Exercise 6: Solution



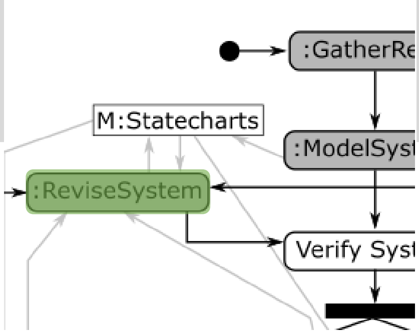
- R8b: when regular operation is resumed the traffic light restarts with the last active light color red (R), green (G), or yellow (Y) on.



Exercise 7

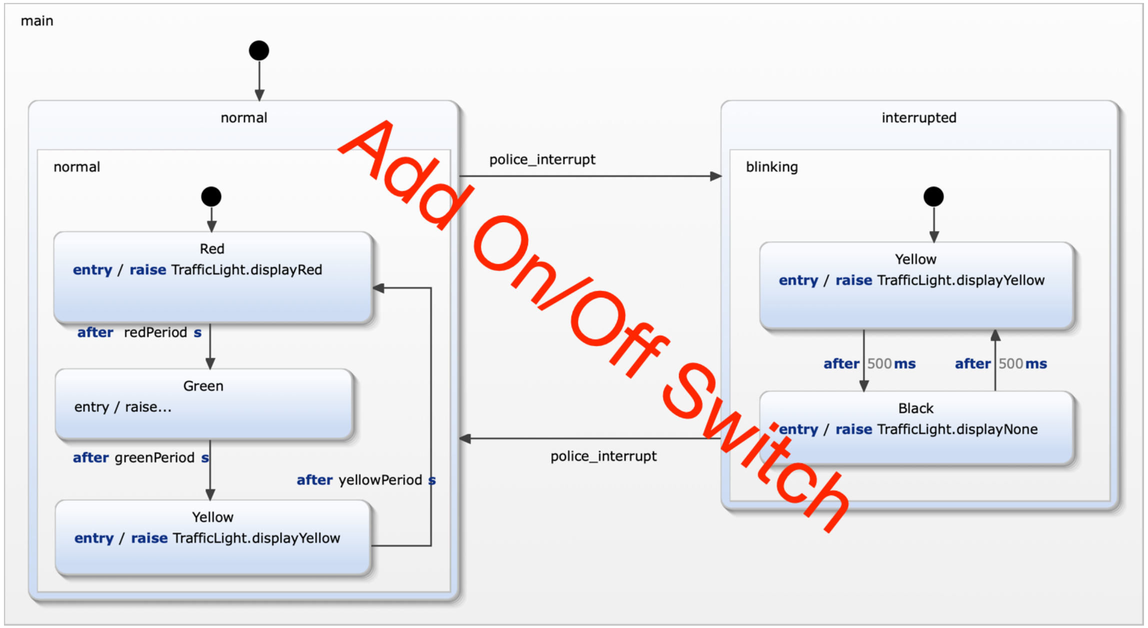
Model an interruptible traffic light that restores its state and can be switched on/off

Exercise 7: Requirements



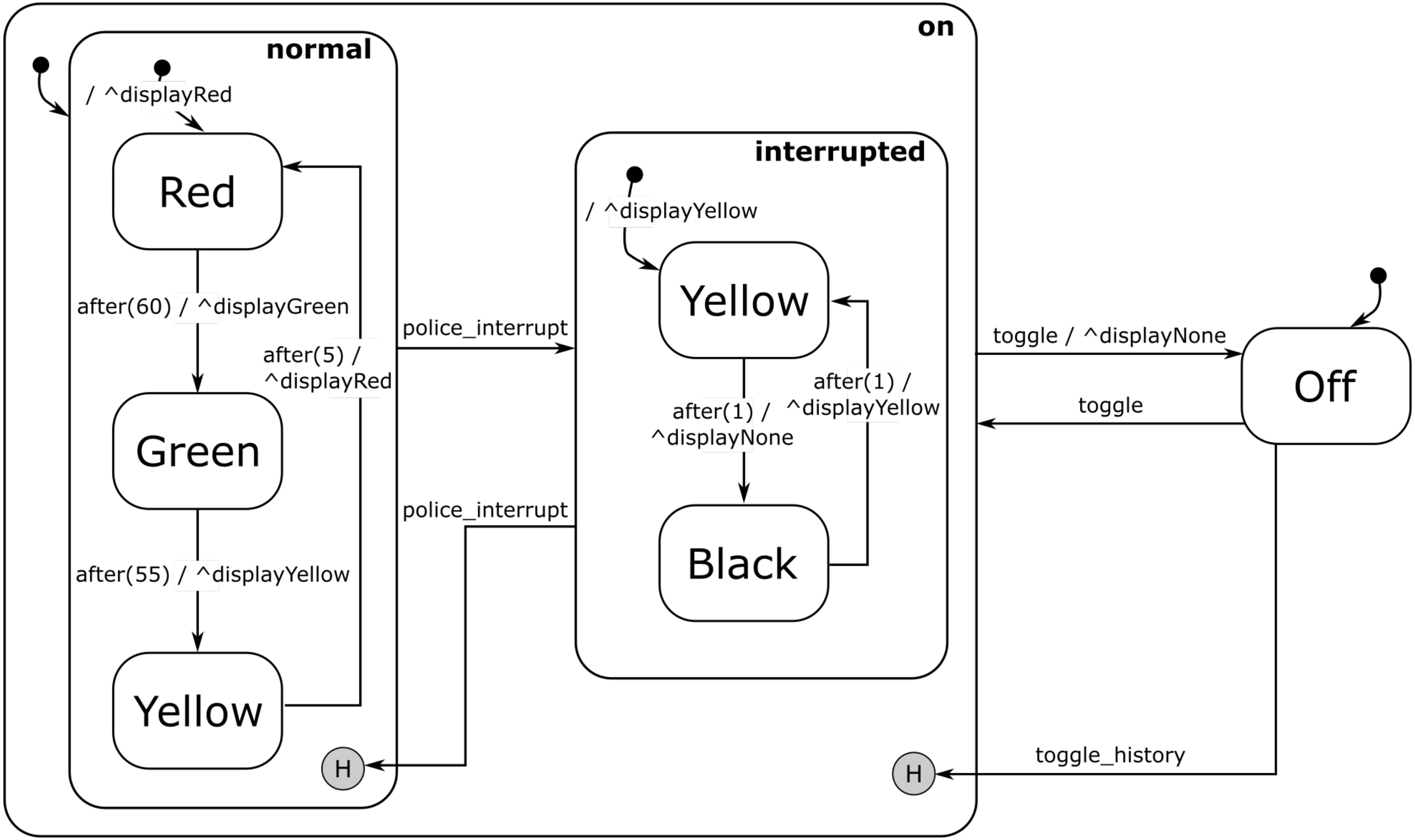
Add another level of hierarchy that supports switching on and off the entire traffic light. Go into detail with shallow and deep histories.

- R9: The traffic light can be switched on and off.
- R9a: The traffic light is initially off.
- R9b: If the traffic light is off none of its lights (R/G/Y) are on.
- R9c: After switching off and on again the traffic light must switch on the light that was on before the switching off.

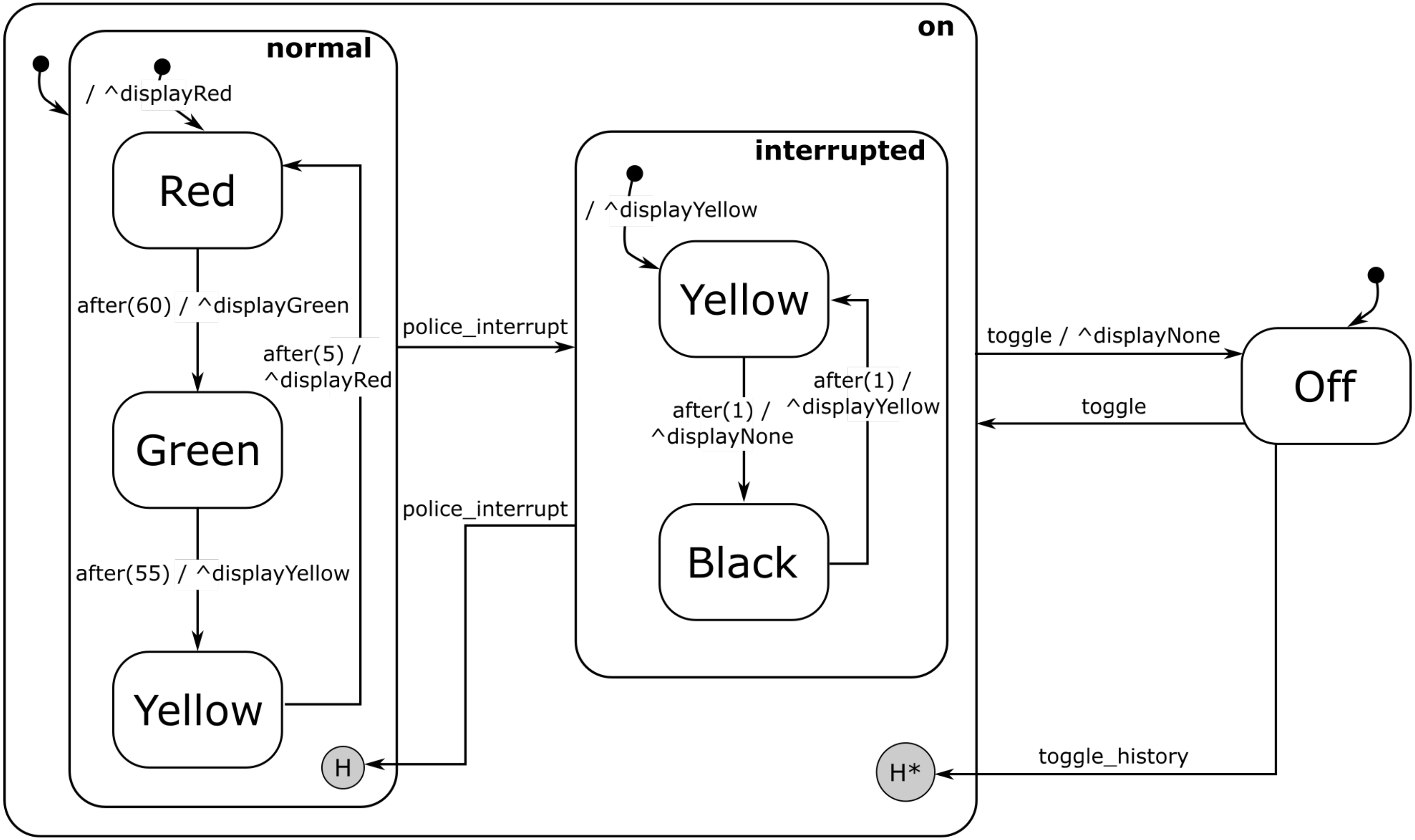


Add On/Off Switch

Exercise 7: Solution



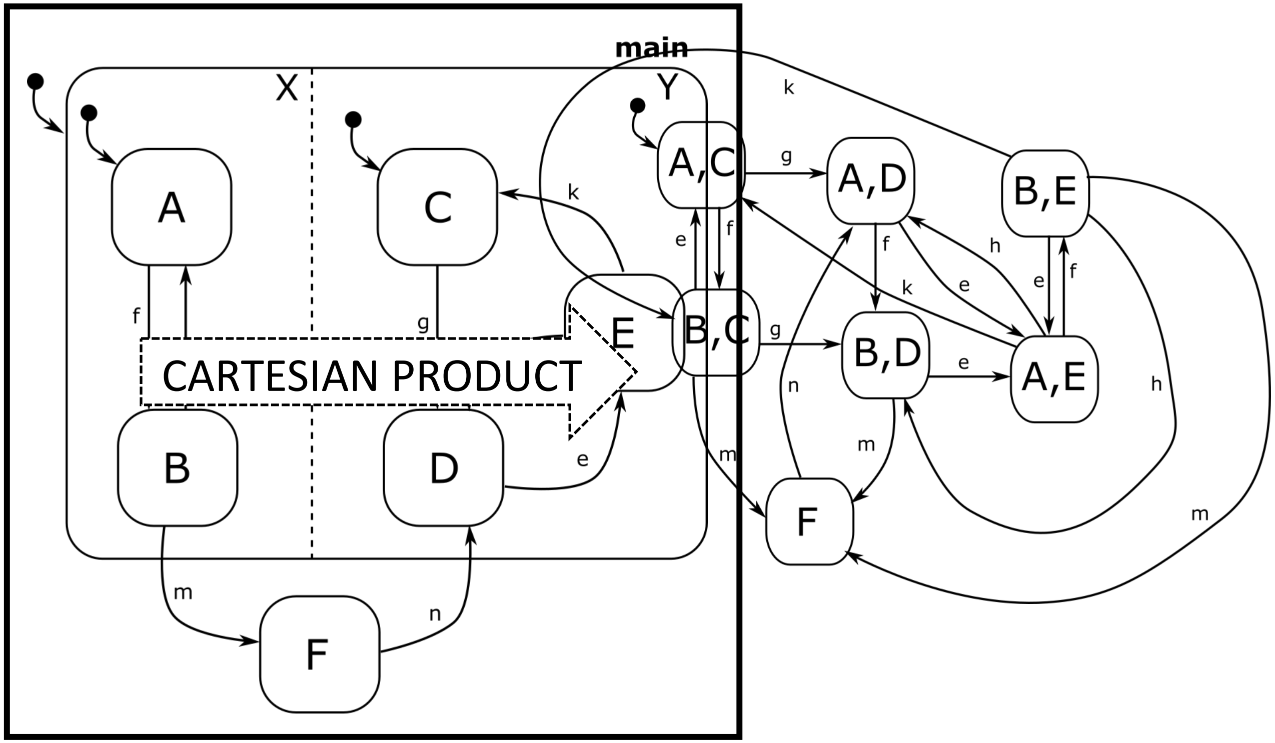
Exercise 7: Alternative Solution



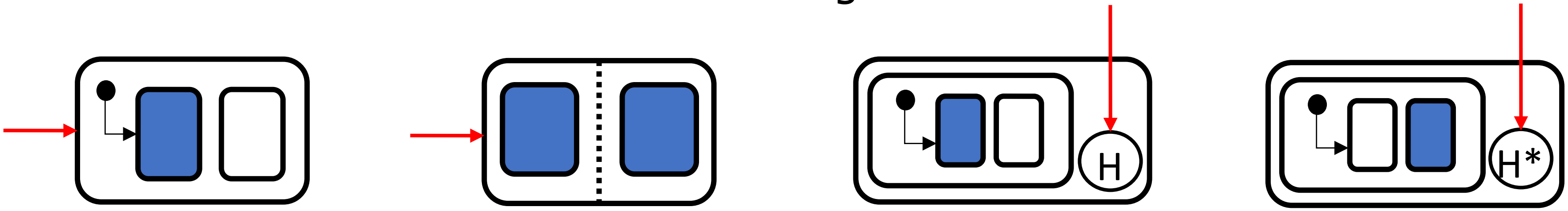
Orthogonality

Orthogonal Components/Regions: “and” states

Semantics/Meaning?

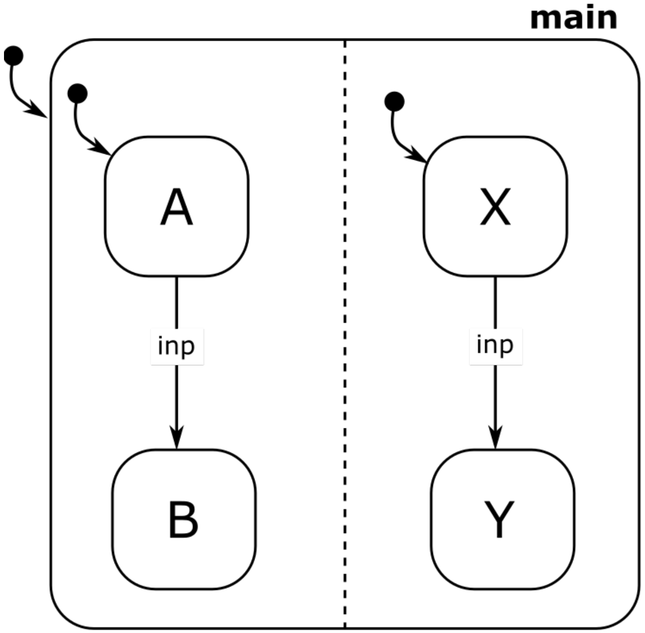


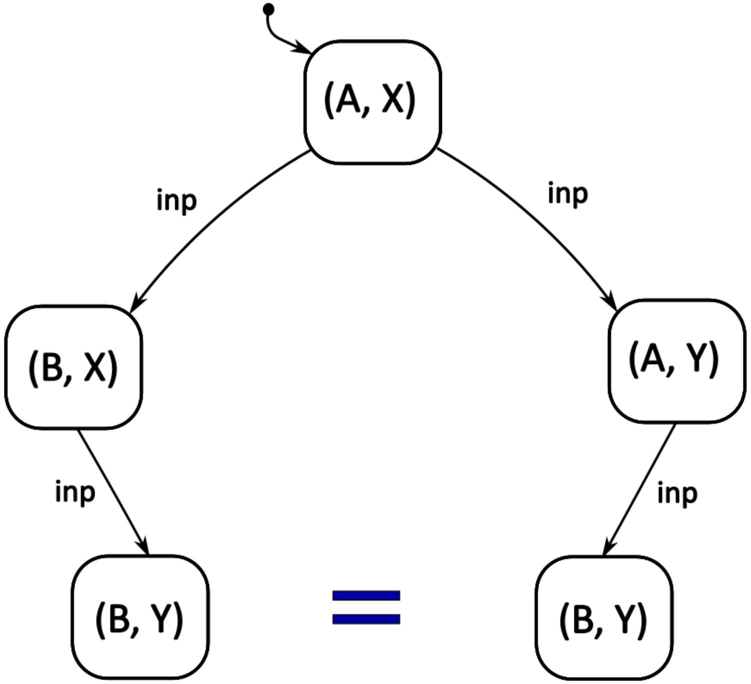
Effective target states:



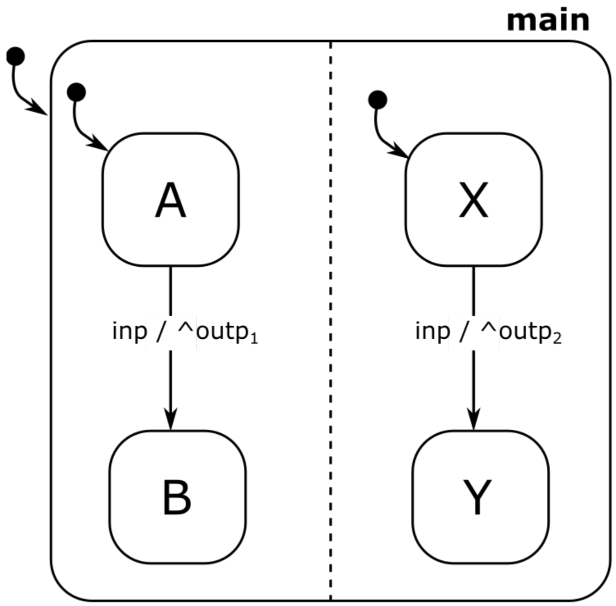
RECURSIVE!

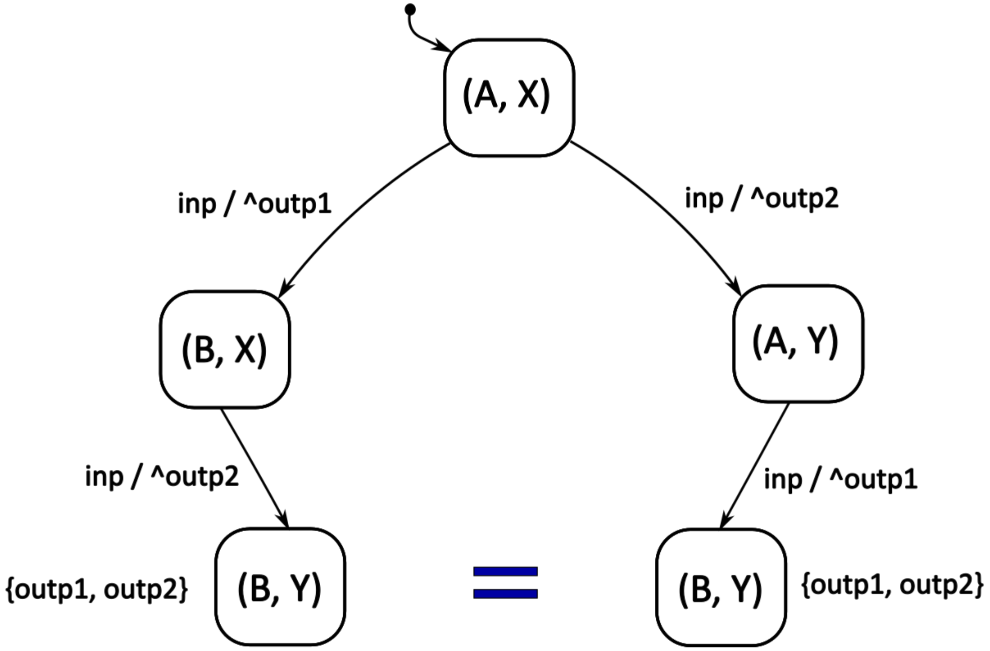
Parallel (In)Dependence



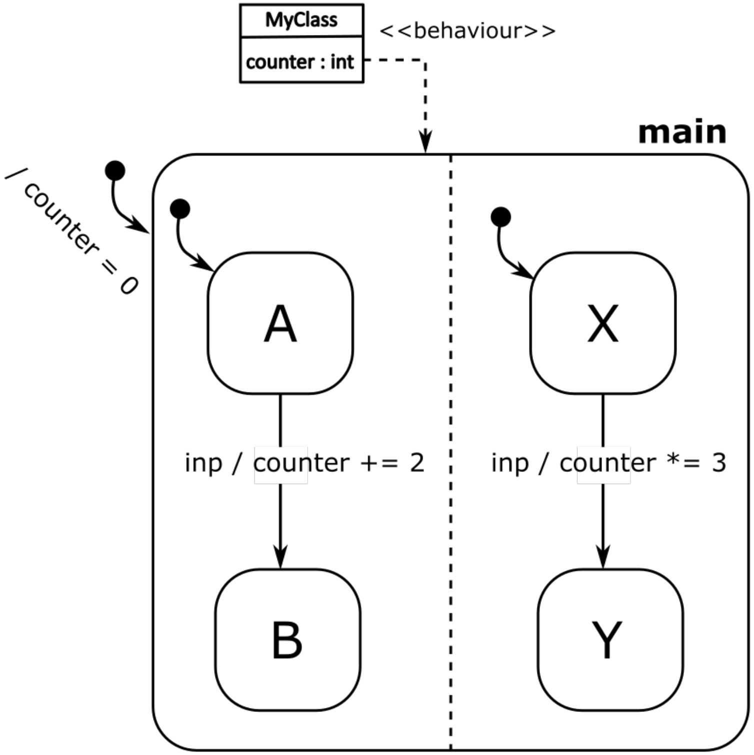


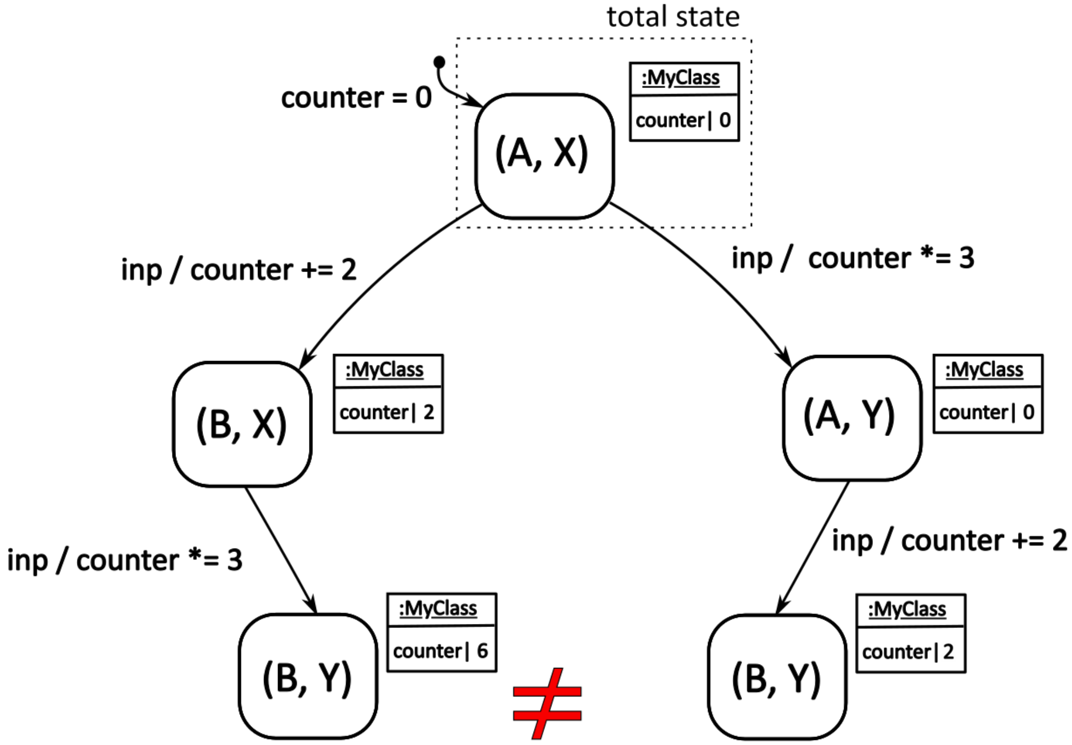
Parallel (In)Dependence



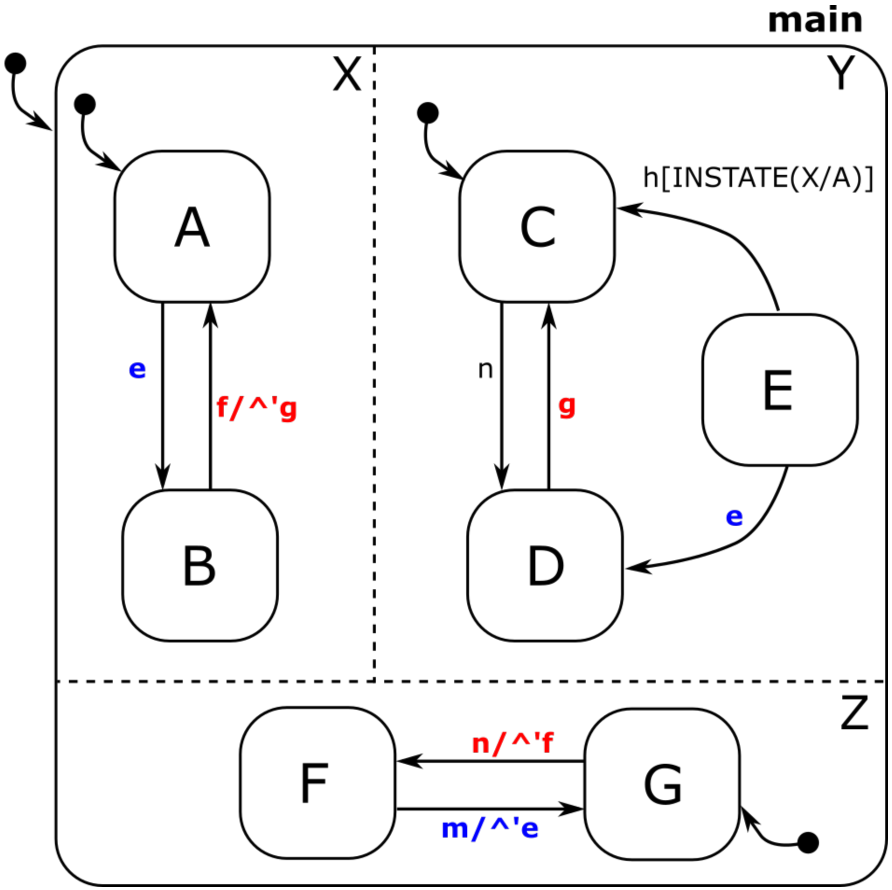


Parallel (In)Dependence





Orthogonality: Communication

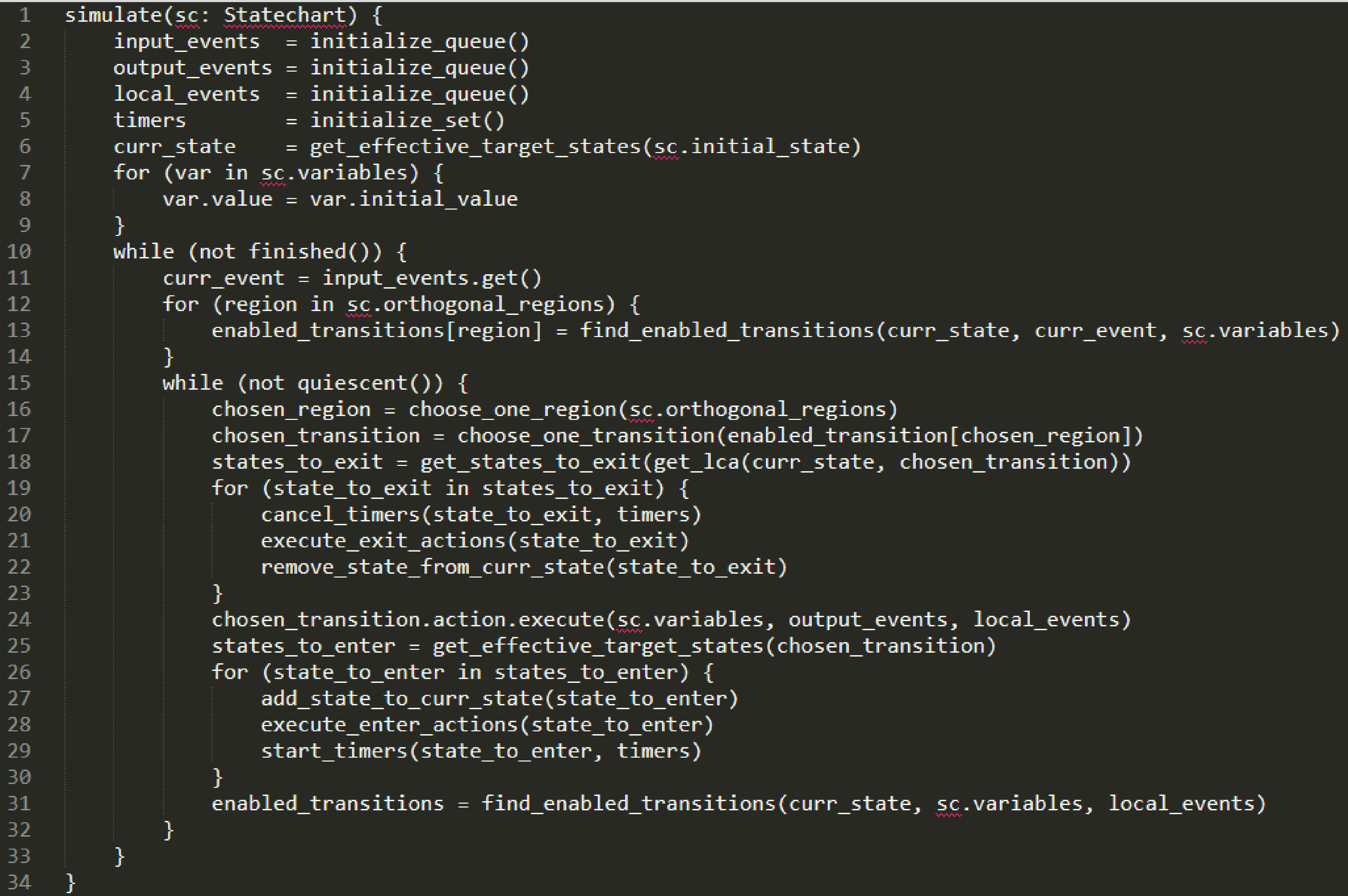


Input Segment: **nmnn**

Orthogonal Components can communicate:

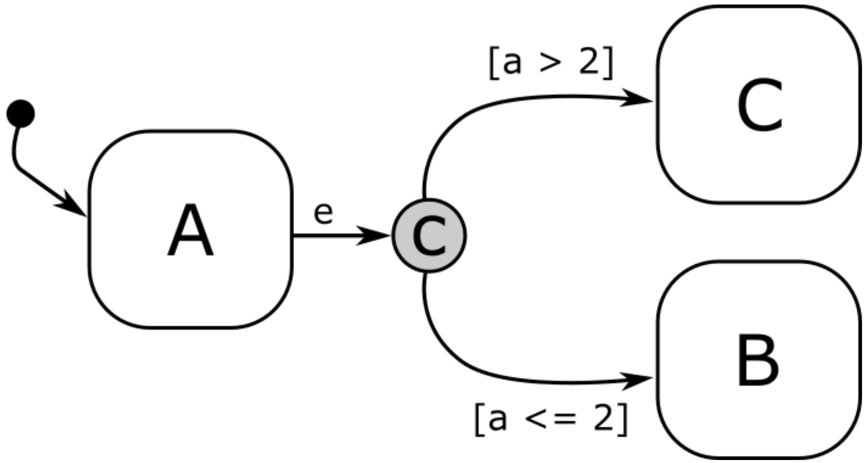
- raising/broadcasting local events:
  ^’<<event name>>
- state reference is a transition guard:
  INSTATE(<<state location>>)

Simulation Algorithm



```
simulate(sc: Statechart) {
    input_events  = initialize_queue()
    output_events = initialize_queue()
    local_events  = initialize_queue()
    timers        = initialize_set()
    curr_state    = get_effective_target_states(sc.initial_state)
    for (var in sc.variables) {
        var.value = var.initial_value
    }
    while (not finished()) {
        curr_event = input_events.get()
        for (region in sc.orthogonal_regions) {
            enabled_transitions[region] = find_enabled_transitions(curr_state, curr_event, sc.variables)
        }
        while (not quiescent()) {
            chosen_region = choose_one_region(sc.orthogonal_regions)
            chosen_transition = choose_one_transition(enabled_transition[chosen_region])
            states_to_exit = get_states_to_exit(get_lca(curr_state, chosen_transition))
            for (state_to_exit in states_to_exit) {
                cancel_timers(state_to_exit, timers)
                execute_exit_actions(state_to_exit)
                remove_state_from_curr_state(state_to_exit)
            }
            chosen_transition.action.execute(sc.variables, output_events, local_events)
            states_to_enter = get_effective_target_states(chosen_transition)
            for (state_to_enter in states_to_enter) {
                add_state_to_curr_state(state_to_enter)
                execute_enter_actions(state_to_enter)
                start_timers(state_to_enter, timers)
            }
            enabled_transitions = find_enabled_transitions(curr_state, sc.variables, local_events)
        }
    }
}
```

Conditional Transitions

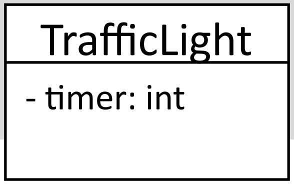


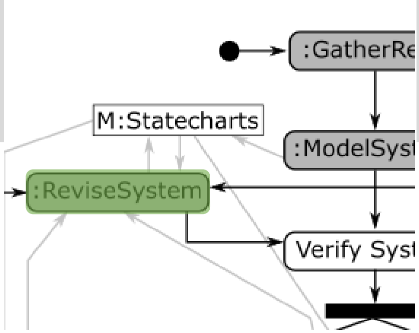
- getEffectiveTargetStates(): select one *True*-branch
- Conditions should not overlap to avoid non-determinism
  - in Yakindu, priority makes deterministic
  - “else” branch is required
- Equivalent (in this case) to two transitions:
  - A – *e[a > 2]* -> C
  - A – *e[a <= 2]* -> B

Exercise 8

Add a timer
to the traffic light

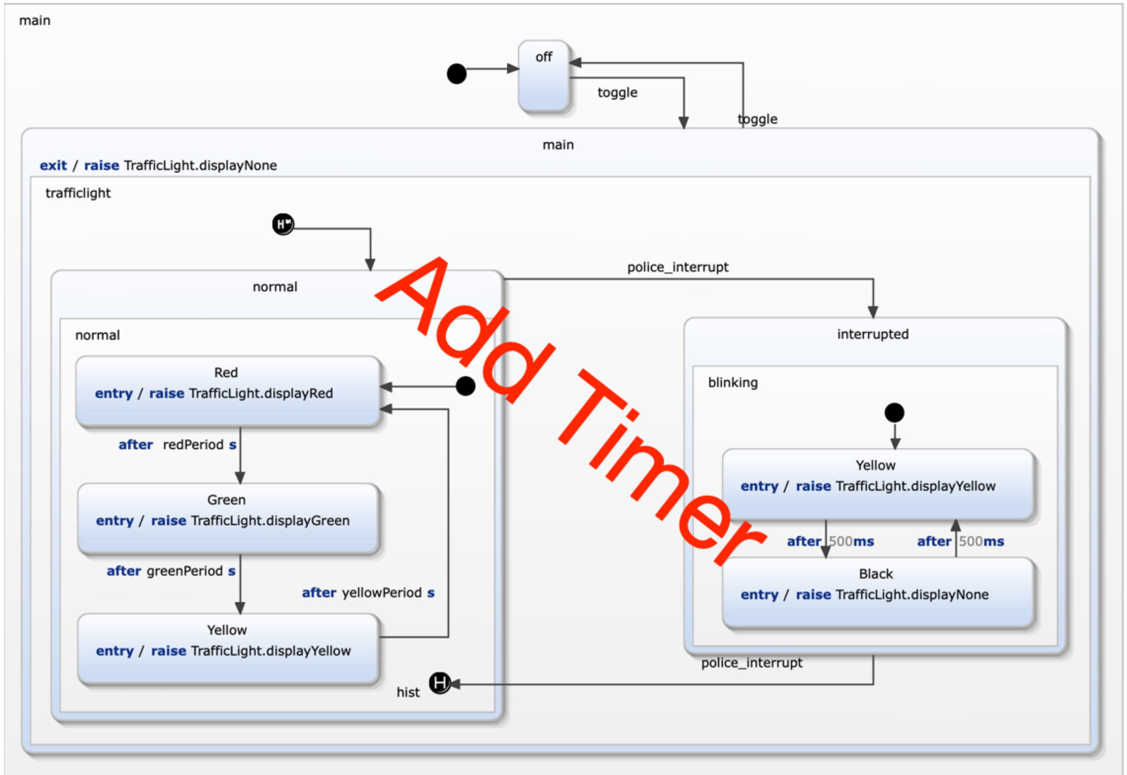
Exercise 8: Requirements





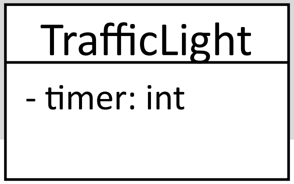
In this exercise a timer must be modelled.
It introduces the use of orthogonal regions.

- R10a: A timer displays the remaining time while the light is red or green.
- R10b: This timer decreases and displays its value every second.
- R10c: The colour of the timer reflects the colour of the traffic light.


main
off
toggle
toggle
main
exit / raise TrafficLight.displayNone
trafficlight
H*
police_interrupt
normal
normal
Red
entry / raise TrafficLight.displayRed
after redPeriod s
Green
entry / raise TrafficLight.displayGreen
after greenPeriod s
after yellowPeriod s
Yellow
entry / raise TrafficLight.displayYellow
hist
H
interrupted
blinking
Yellow
entry / raise TrafficLight.displayYellow
after 500ms
after 500ms
Black
entry / raise TrafficLight.displayNone
police_interrupt

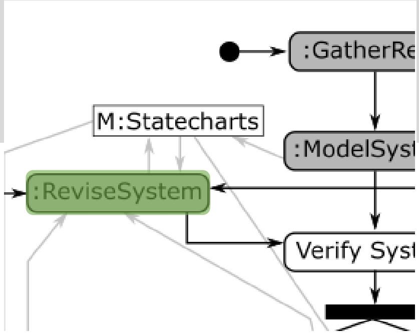
Add Timer

Exercise 8: Solution

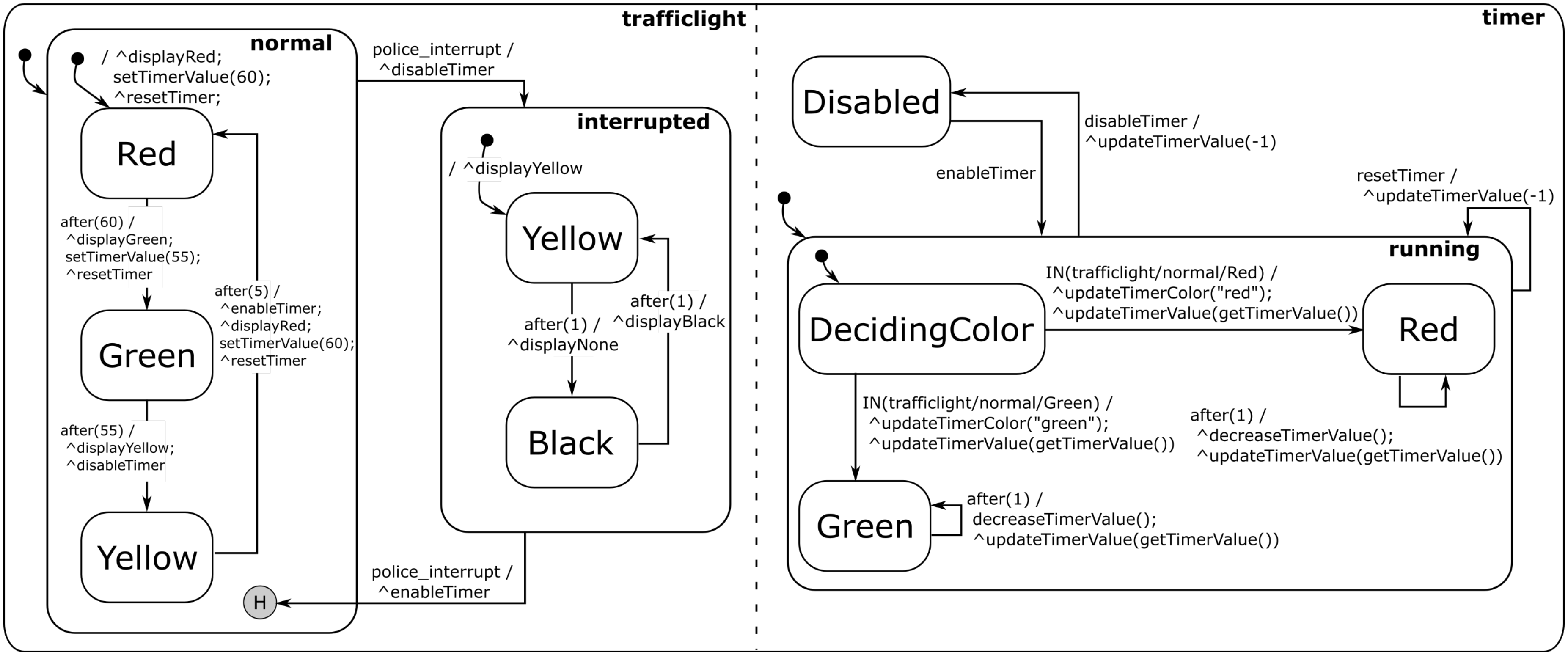


TrafficLight

- timer: int



- R10a: A timer displays the remaining time while the light is red or green.
- R10b: This timer decreases and displays its value every second.
- R10c: The colour of the timer reflects the colour of the traffic light.



trafficlight

normal

/ ^displayRed;
setTimerValue(60);
^resetTimer;

Red

after(60) /
^displayGreen;
setTimerValue(55);
^resetTimer

Green

after(55) /
^displayYellow;
^disableTimer

Yellow

after(5) /
^enableTimer;
^displayRed;
setTimerValue(60);
^resetTimer

H

police_interrupt /
^disableTimer

interrupted

/ ^displayYellow

Yellow

after(1) /
^displayNone

Black

after(1) /
^displayBlack

police_interrupt /
^enableTimer

timer

Disabled

disableTimer /
^updateTimerValue(-1)

enableTimer

resetTimer /
^updateTimerValue(-1)

running

DecidingColor

IN(trafficlight/normal/Red) /
^updateTimerColor("red");
^updateTimerValue(getTimerValue())

Red

IN(trafficlight/normal/Green) /
^updateTimerColor("green");
^updateTimerValue(getTimerValue())

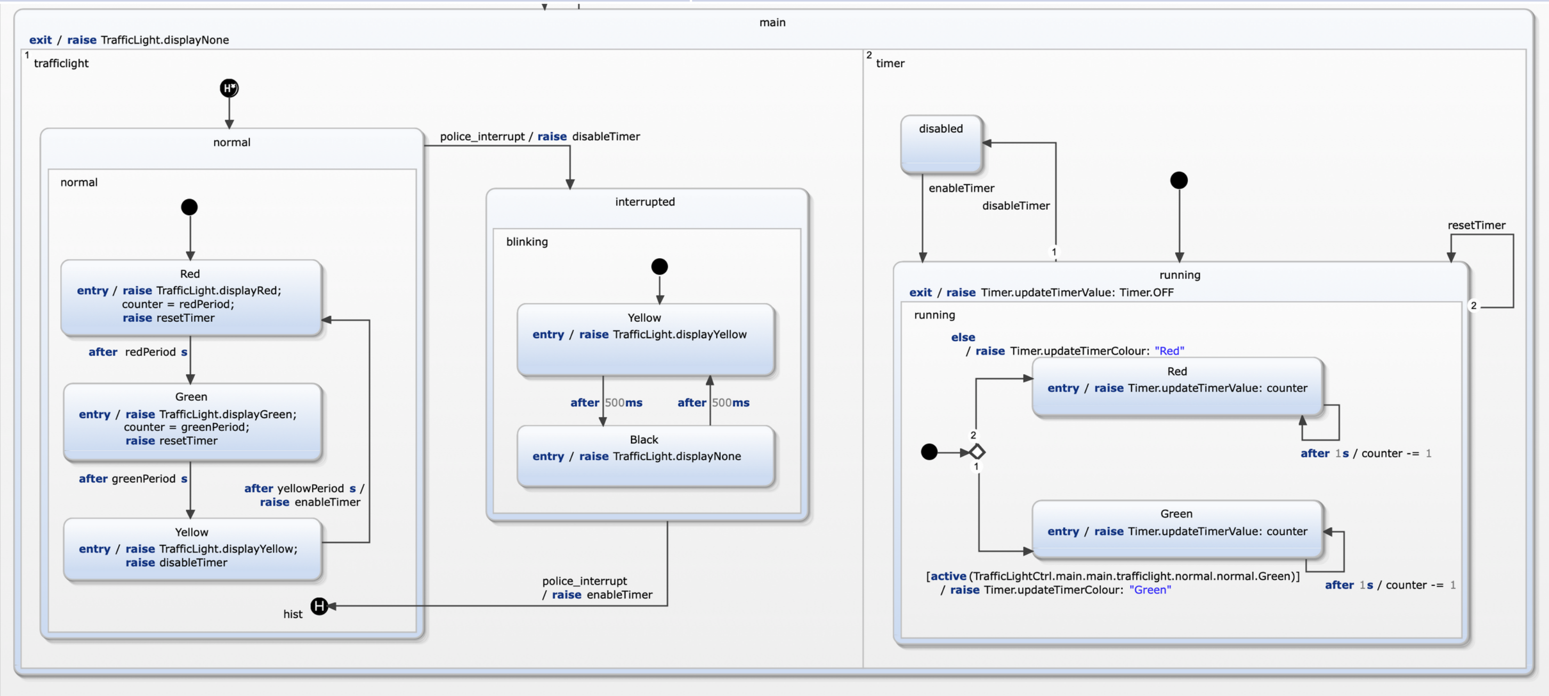
after(1) /
^decreaseTimerValue();
^updateTimerValue(getTimerValue())

Green

after(1) /
decreaseTimerValue();
^updateTimerValue(getTimerValue())

Solution 8

requirement	modelling approach
R10: a timer displays the remaining time while the light is red or green	The timer is defined in a second region within state on (main in the Yakindu model).
R10a: This timer decreases and displays its value every second.	An internal variable for the counter is introduced. When switching the traffic light phase, the counter value is set to how long the light has been in that phase. Additionally, the local events resetTimer, enableTimer, and disableTimer are used to synchronize traffic light phase switches with the timer.
R10b: The colour of the timer reflects the colour of the traffic light.	When the timer is enabled it checks the active traffic light phase using the active() function.


main
exit / raise TrafficLight.displayNone
1 trafficlight
H*
normal
normal
Red
entry / raise TrafficLight.displayRed;
counter = redPeriod;
raise resetTimer
after redPeriod s
Green
entry / raise TrafficLight.displayGreen;
counter = greenPeriod;
raise resetTimer
after greenPeriod s
after yellowPeriod s /
raise enableTimer
Yellow
entry / raise TrafficLight.displayYellow;
raise disableTimer
hist
H
police_interrupt / raise disableTimer
interrupted
blinking
Yellow
entry / raise TrafficLight.displayYellow
after 500ms
after 500ms
Black
entry / raise TrafficLight.displayNone
police_interrupt
/ raise enableTimer
2 timer
disabled
enableTimer
disableTimer
1
resetTimer
running
exit / raise Timer.updateTimerValue: Timer.OFF
2
running
else
/ raise Timer.updateTimerColour: "Red"
Red
entry / raise Timer.updateTimerValue: counter
after 1s / counter -= 1
2
1
Green
entry / raise Timer.updateTimerValue: counter
[active (TrafficLightCtrl.main.main.trafficlight.normal.normal.Green)]
/ raise Timer.updateTimerColour: "Green"
after 1s / counter -= 1

Yakindu syntax

Yakindu:

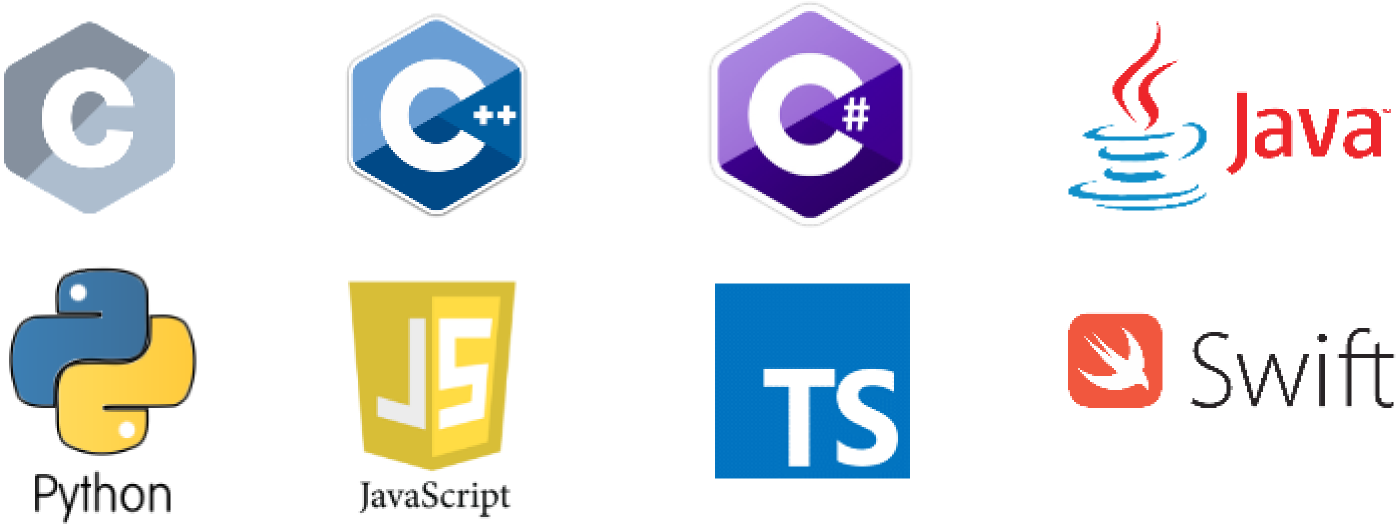
- **raise** e == ^e
- strict alternation between “or” and “and” states

  TrafficLightCtrl.**main.main**.trafficlight.**normal.normal**.Green
- **active**() == INSTATE() == IN()

Code Generation

Code Generation

- Code generators for C, C++, Java, Python, Swift, Typescript, SCXML
- Plain-code approach by default
- Very efficient code
- Easy integration of custom generators



Code Generation

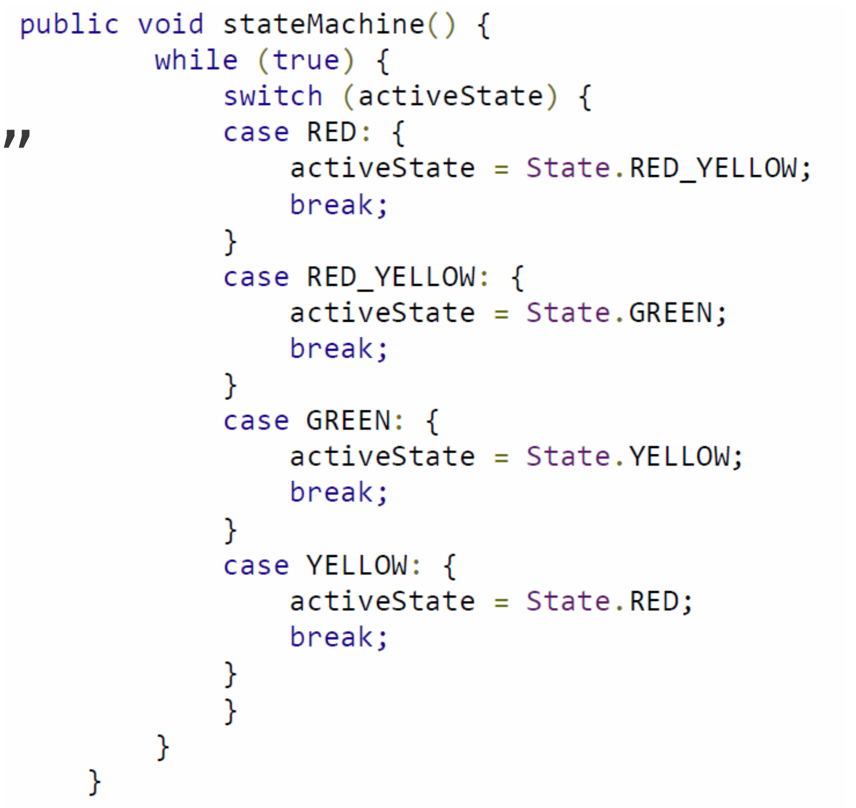
- Various different approaches for implementing a state machine:
  - switch / case
  - state transition table
  - state pattern

- Which one is the best depends on
  - Runtime (performance, predictability) requirements
  - ROM vs. RAM memory
  - Debugging capabilities
  - Clarity and maintainability
    (of generated code ~ certification, round-trip)

Switch / Case

- Each state corresponds to one “case”

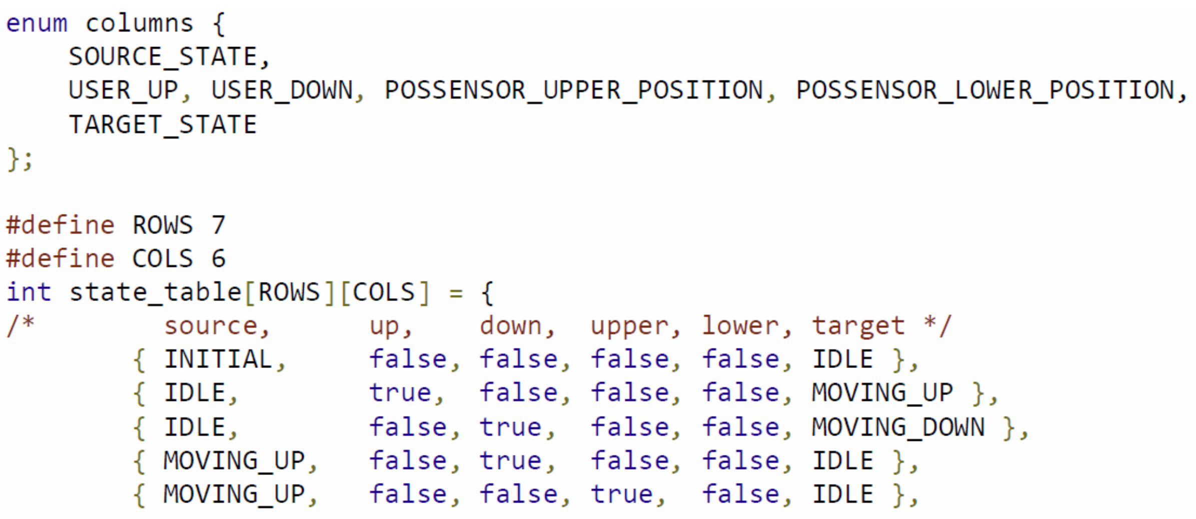
- Each case executes state-specific statements and state transitions



```
public void stateMachine() {
        while (true) {
            switch (activeState) {
            case RED: {
                activeState = State.RED_YELLOW;
                break;
            }
            case RED_YELLOW: {
                activeState = State.GREEN;
                break;
            }
            case GREEN: {
                activeState = State.YELLOW;
                break;
            }
            case YELLOW: {
                activeState = State.RED;
                break;
            }
            }
        }
    }
```

State Transition Table

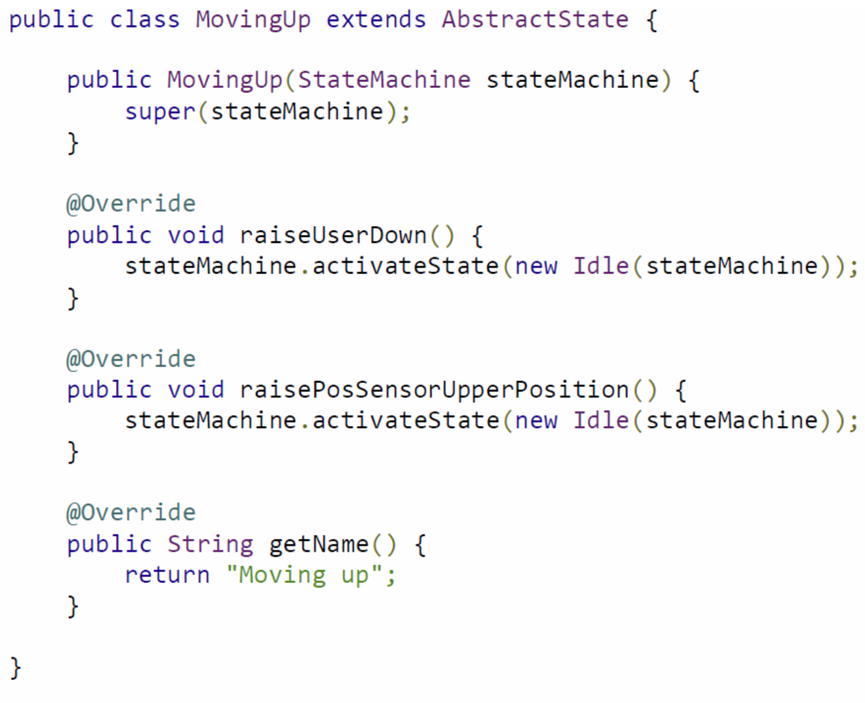
- Specifies the state machine purely declaratively.
- One of the dimensions indicates current states, while the other indicates events.



```
enum columns {
    SOURCE_STATE,
    USER_UP, USER_DOWN, POSSENSOR_UPPER_POSITION, POSSENSOR_LOWER_POSITION,
    TARGET_STATE
};

#define ROWS 7
#define COLS 6
int state_table[ROWS][COLS] = {
/*          source,      up,    down,   upper, lower, target */
        { INITIAL,     false, false, false, false, IDLE },
        { IDLE,        true,  false, false, false, MOVING_UP },
        { IDLE,        false, true,  false, false, MOVING_DOWN },
        { MOVING_UP,   false, true,  false, false, IDLE },
        { MOVING_UP,   false, false, true,  false, IDLE },
```

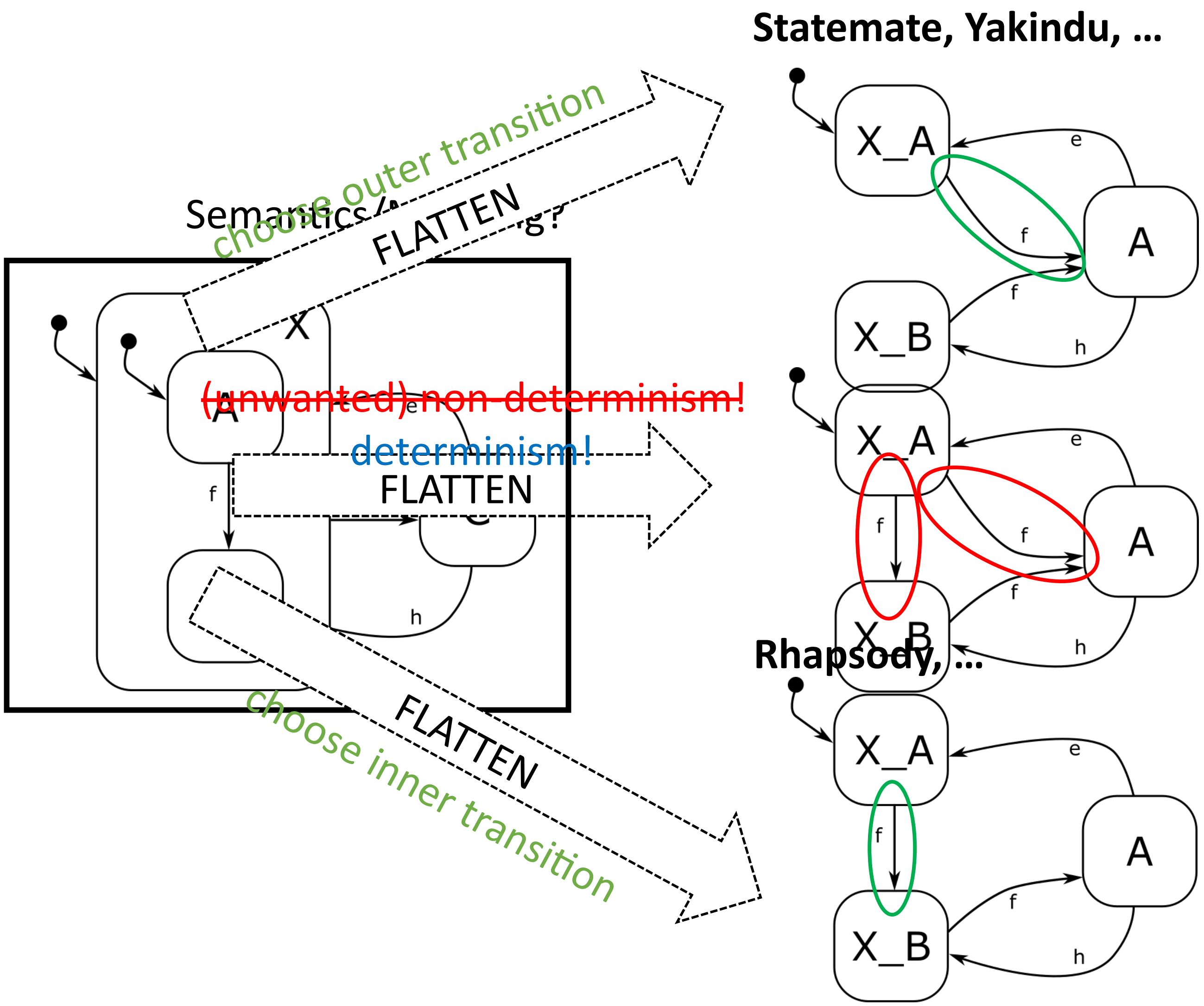
State Pattern

- Object-oriented implementation, behavioural design pattern
- Used by frameworks such as Spring Statemachine, Boost MSM or Qt State Machine Framework
- Each State becomes a class, events become methods
- All classes derive from a common interface



```
public class MovingUp extends AbstractState {

    public MovingUp(StateMachine stateMachine) {
        super(stateMachine);
    }

    @Override
    public void raiseUserDown() {
        stateMachine.activateState(new Idle(stateMachine));
    }

    @Override
    public void raisePosSensorUpperPosition() {
        stateMachine.activateState(new Idle(stateMachine));
    }

    @Override
    public String getName() {
        return "Moving up";
    }

}
```

Hierarchy: outer vs. inner



Statemate, Yakindu, …

Semantics/Meaning?

choose outer transition

FLATTEN

~~(unwanted) non-determinism!~~

determinism!

FLATTEN

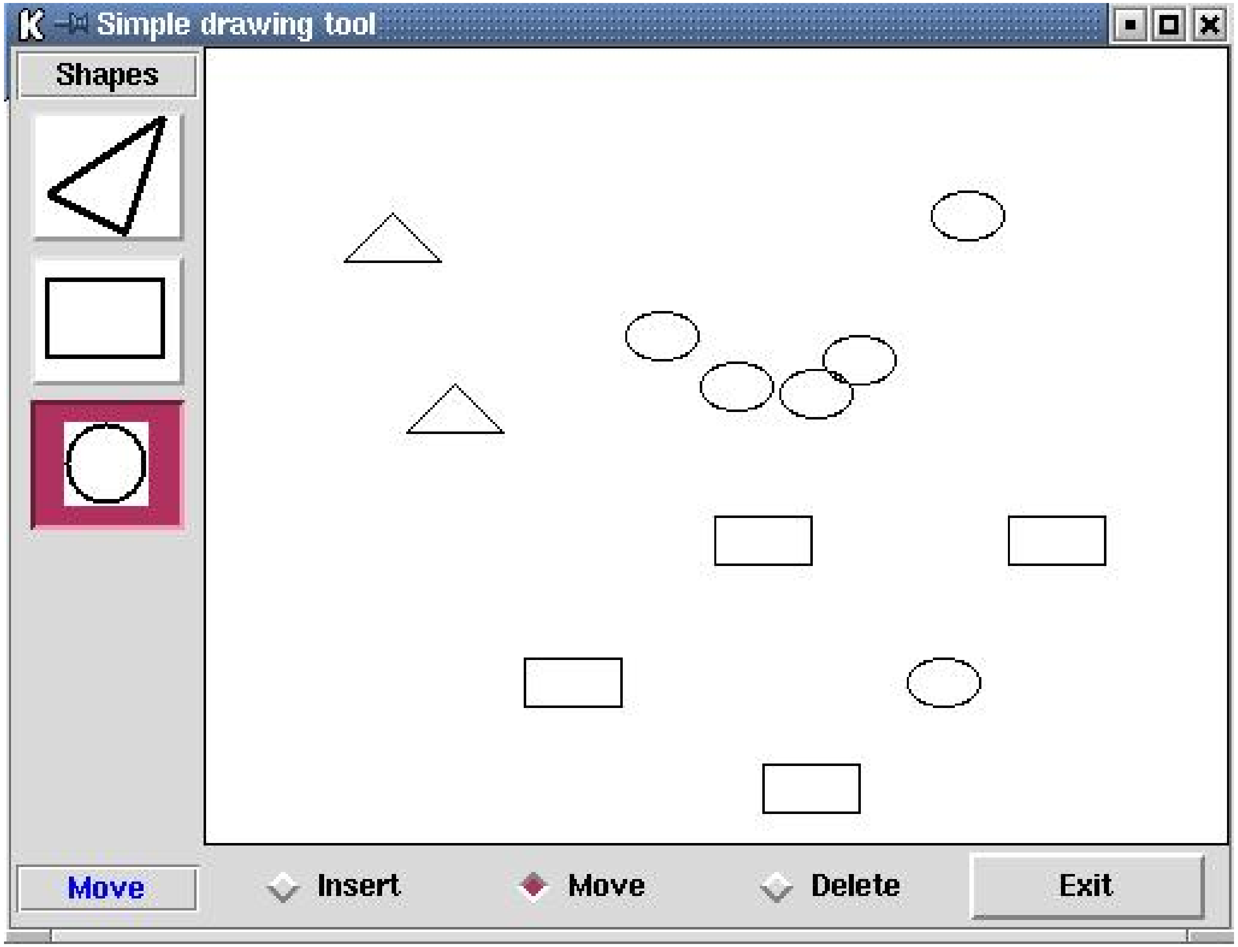
FLATTEN

choose inner transition

Rhapsody, …

Hierarchy: why inner?

Manual Object-Oriented implementation (no Statechart compiler) using State Pattern



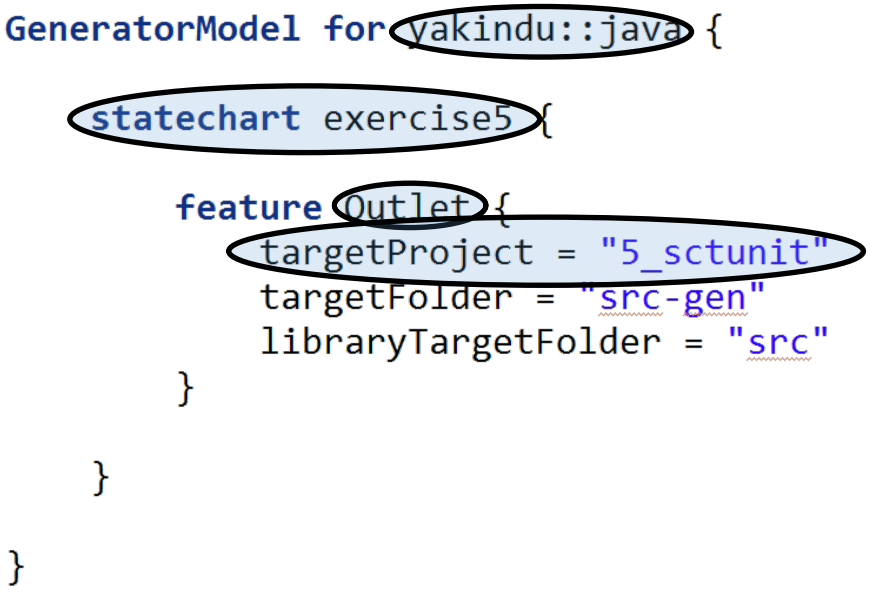
http://msdl.cs.mcgill.ca/people/hv/teaching/SoftwareDesign/COMP304B2002/lectures/lecture.statecharts/

Code Generation

		Fast	Memory efficient	easy to debug	Easy to understand
SCT	Switch / Case	+	+	0	0
	State Transition Table	+	0	−	−
	State Pattern	−	−	+	+

⚠ very simplified illustration

Code Generator Model



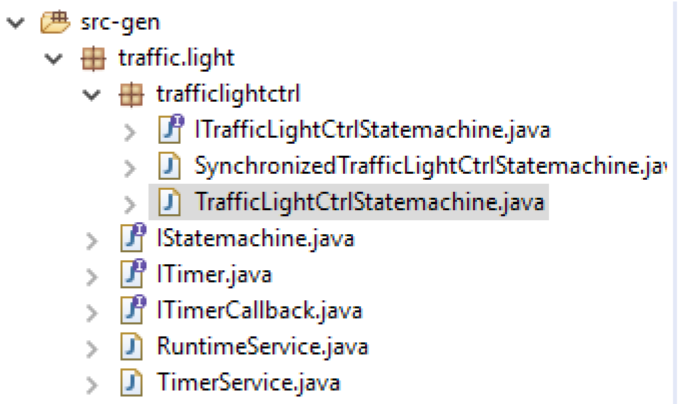
```
GeneratorModel for yakindu::java {

    statechart exercise5 {

        feature Outlet {
            targetProject = "5_sctunit"
            targetFolder = "src-gen"
            libraryTargetFolder = "src"
        }

    }

}
```

- Has a generator ID
- Has a generator entry
- Each generator entry contains 1..n feature-configurations
- Each feature-configuration contains 1..n properties

Generated Code

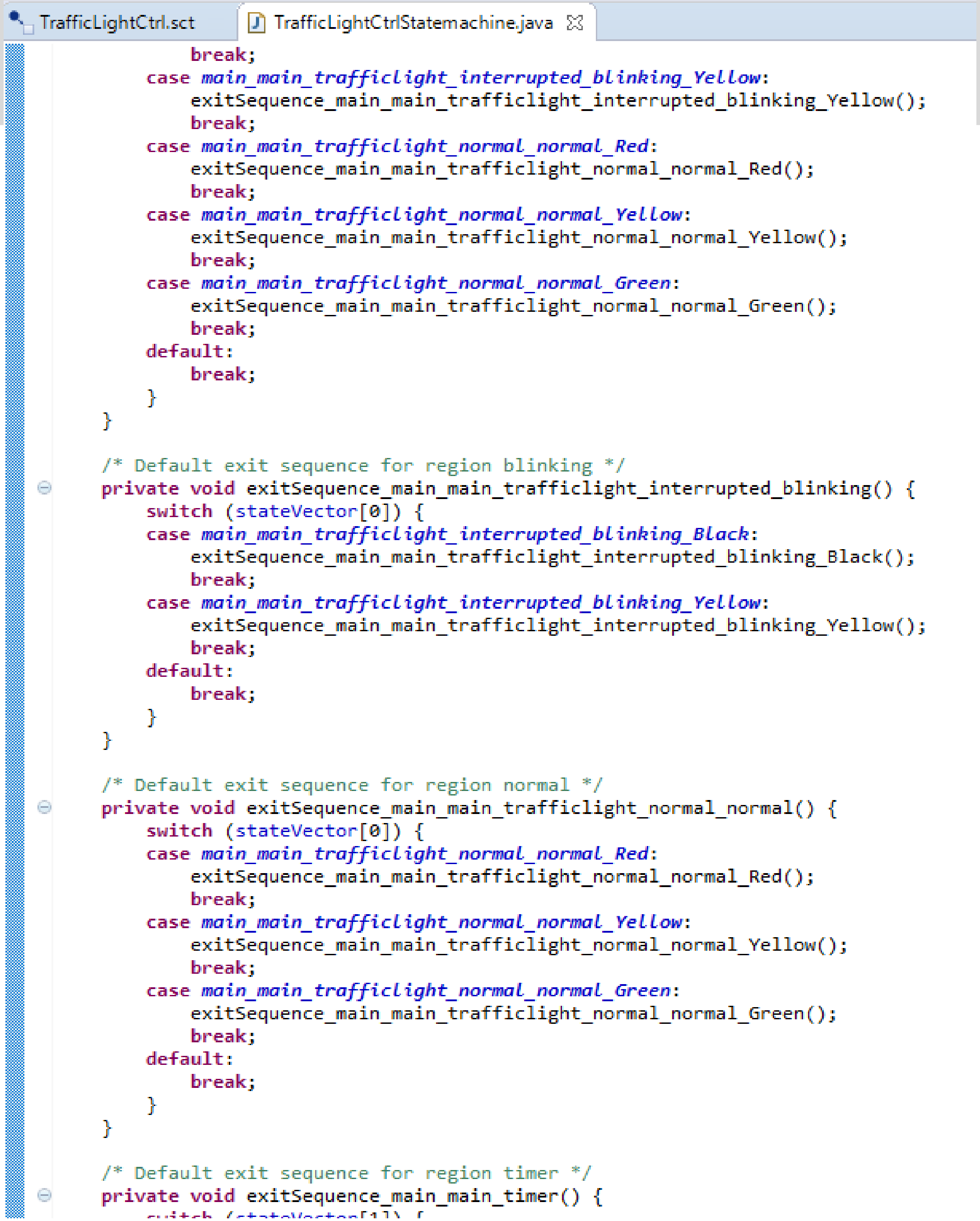
Sample

Files



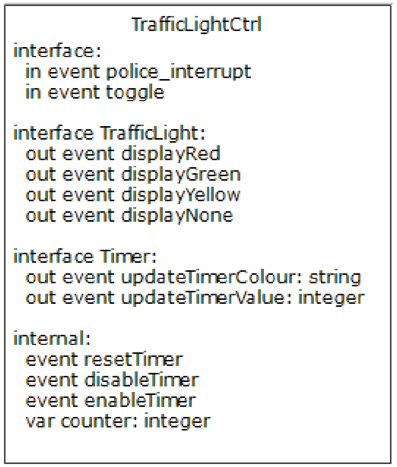
```
src-gen
    traffic.light
        trafficlightctrl
            ITrafficLightCtrlStatemachine.java
            SynchronizedTrafficLightCtrlStatemachine.ja
            TrafficLightCtrlStatemachine.java
        IStatemachine.java
        ITimer.java
        ITimerCallback.java
        RuntimeService.java
        TimerService.java
```

- 8 files
- 1311 lines of code
- 302 manual (UI) code



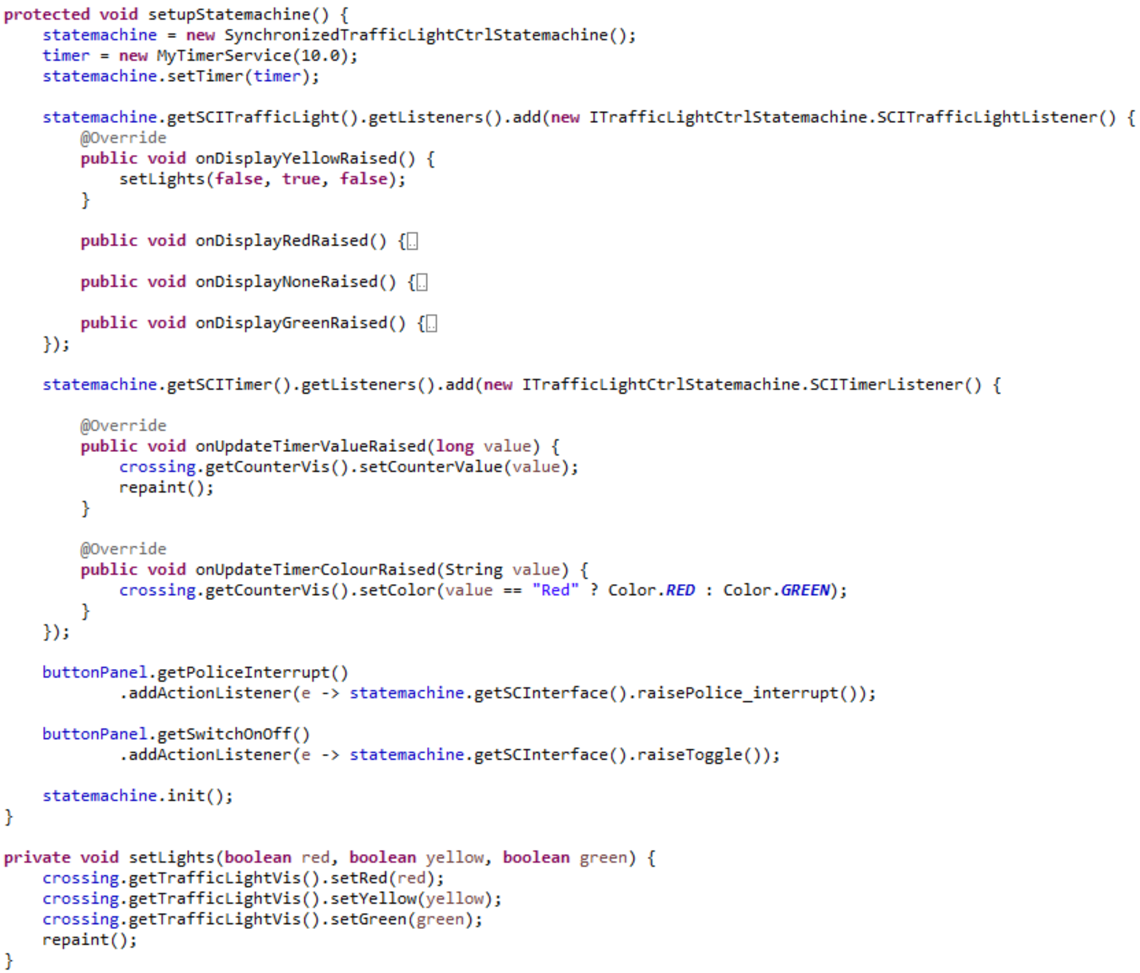
```java
            break;
        case main_main_trafficlight_interrupted_blinking_Yellow:
            exitSequence_main_main_trafficlight_interrupted_blinking_Yellow();
            break;
        case main_main_trafficlight_normal_normal_Red:
            exitSequence_main_main_trafficlight_normal_normal_Red();
            break;
        case main_main_trafficlight_normal_normal_Yellow:
            exitSequence_main_main_trafficlight_normal_normal_Yellow();
            break;
        case main_main_trafficlight_normal_normal_Green:
            exitSequence_main_main_trafficlight_normal_normal_Green();
            break;
        default:
            break;
        }
    }

    /* Default exit sequence for region blinking */
    private void exitSequence_main_main_trafficlight_interrupted_blinking() {
        switch (stateVector[0]) {
        case main_main_trafficlight_interrupted_blinking_Black:
            exitSequence_main_main_trafficlight_interrupted_blinking_Black();
            break;
        case main_main_trafficlight_interrupted_blinking_Yellow:
            exitSequence_main_main_trafficlight_interrupted_blinking_Yellow();
            break;
        default:
            break;
        }
    }

    /* Default exit sequence for region normal */
    private void exitSequence_main_main_trafficlight_normal_normal() {
        switch (stateVector[0]) {
        case main_main_trafficlight_normal_normal_Red:
            exitSequence_main_main_trafficlight_normal_normal_Red();
            break;
        case main_main_trafficlight_normal_normal_Yellow:
            exitSequence_main_main_trafficlight_normal_normal_Yellow();
            break;
        case main_main_trafficlight_normal_normal_Green:
            exitSequence_main_main_trafficlight_normal_normal_Green();
            break;
        default:
            break;
        }
    }

    /* Default exit sequence for region timer */
    private void exitSequence_main_main_timer() {
        switch (stateVector[1]) {
```

Interface



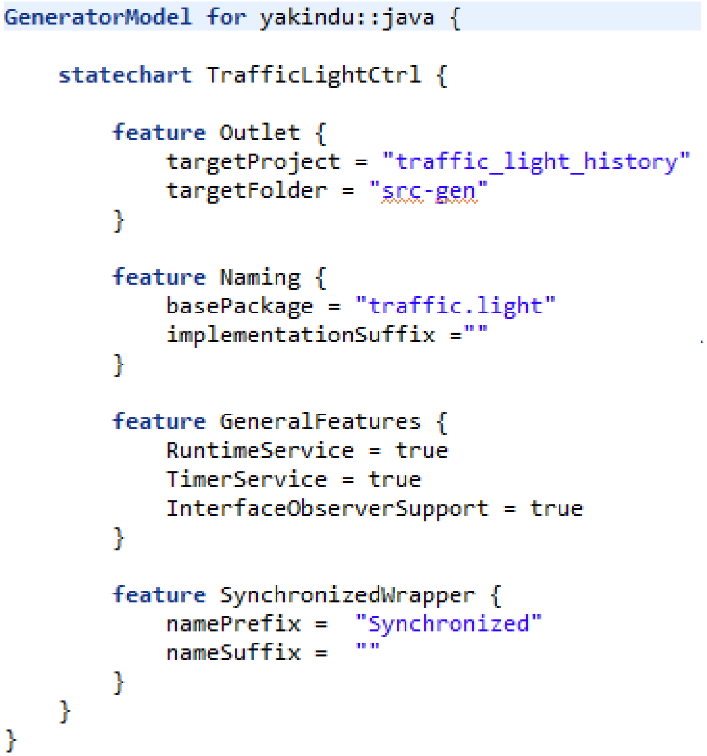
```
TrafficLightCtrl
interface:
  in event police_interrupt
  in event toggle

interface TrafficLight:
  out event displayRed
  out event displayGreen
  out event displayYellow
  out event displayNone

interface Timer:
  out event updateTimerColour: string
  out event updateTimerValue: integer

internal:
  event resetTimer
  event disableTimer
  event enableTimer
  var counter: integer
```

Setup Code (Excerpt)



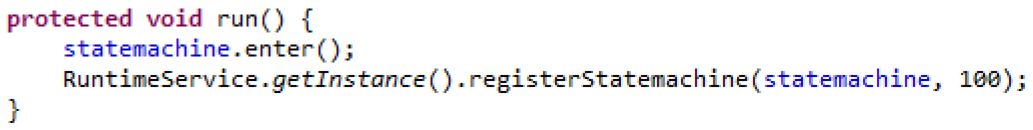
```java
protected void setupStatemachine() {
    statemachine = new SynchronizedTrafficLightCtrlStatemachine();
    timer = new MyTimerService(10.0);
    statemachine.setTimer(timer);

    statemachine.getSCITrafficLight().getListeners().add(new ITrafficLightCtrlStatemachine.SCITrafficLightListener() {
        @Override
        public void onDisplayYellowRaised() {
            setLights(false, true, false);
        }

        public void onDisplayRedRaised() {□

        public void onDisplayNoneRaised() {□

        public void onDisplayGreenRaised() {□
    });

    statemachine.getSCITimer().getListeners().add(new ITrafficLightCtrlStatemachine.SCITimerListener() {

        @Override
        public void onUpdateTimerValueRaised(long value) {
            crossing.getCounterVis().setCounterValue(value);
            repaint();
        }

        @Override
        public void onUpdateTimerColourRaised(String value) {
            crossing.getCounterVis().setColor(value == "Red" ? Color.RED : Color.GREEN);
        }
    });

    buttonPanel.getPoliceInterrupt()
            .addActionListener(e -> statemachine.getSCInterface().raisePolice_interrupt());

    buttonPanel.getSwitchOnOff()
            .addActionListener(e -> statemachine.getSCInterface().raiseToggle());

    statemachine.init();
}

private void setLights(boolean red, boolean yellow, boolean green) {
    crossing.getTrafficLightVis().setRed(red);
    crossing.getTrafficLightVis().setYellow(yellow);
    crossing.getTrafficLightVis().setGreen(green);
    repaint();
}
```

Generator



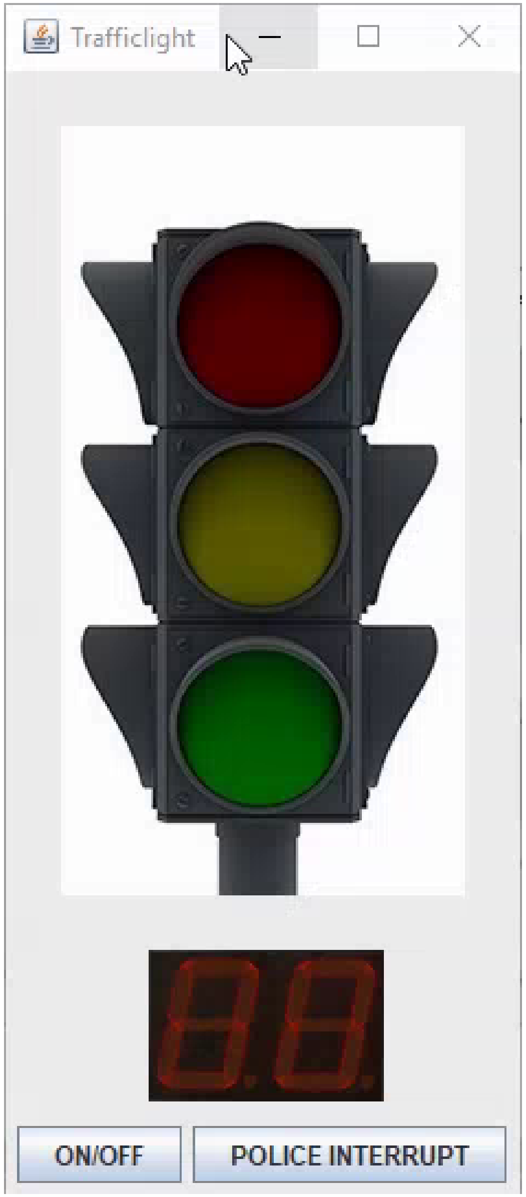
```
GeneratorModel for yakindu::java {

    statechart TrafficLightCtrl {

        feature Outlet {
            targetProject = "traffic_light_history"
            targetFolder = "src-gen"
        }

        feature Naming {
            basePackage = "traffic.light"
            implementationSuffix =""
        }

        feature GeneralFeatures {
            RuntimeService = true
            TimerService = true
            InterfaceObserverSupport = true
        }

        feature SynchronizedWrapper {
            namePrefix =  "Synchronized"
            nameSuffix =  ""
        }
    }
}
```

Runner



```java
protected void run() {
    statemachine.enter();
    RuntimeService.getInstance().registerStatemachine(statemachine, 100);
}
```

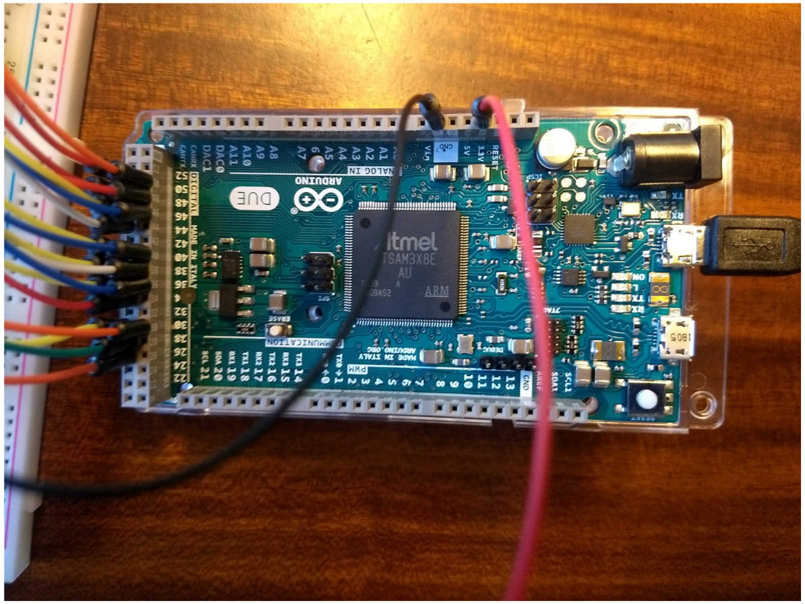
Deployed Application (Scaled Real-Time)

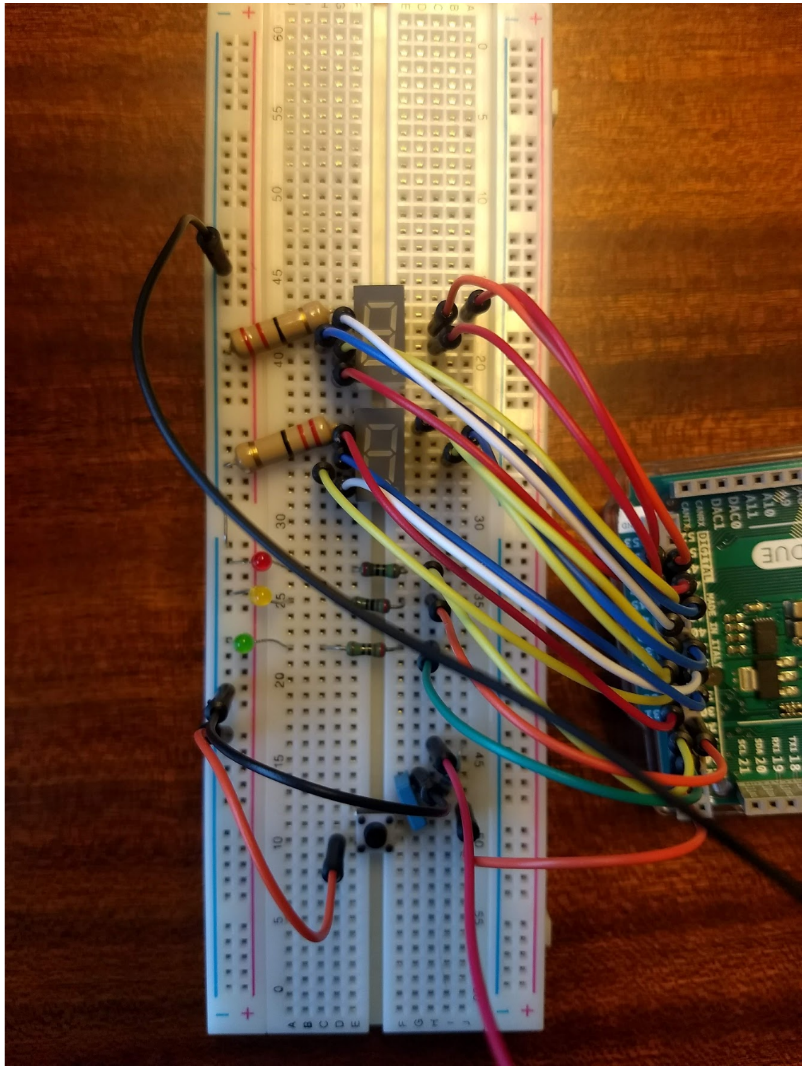


Deploying onto Hardware

Interface:

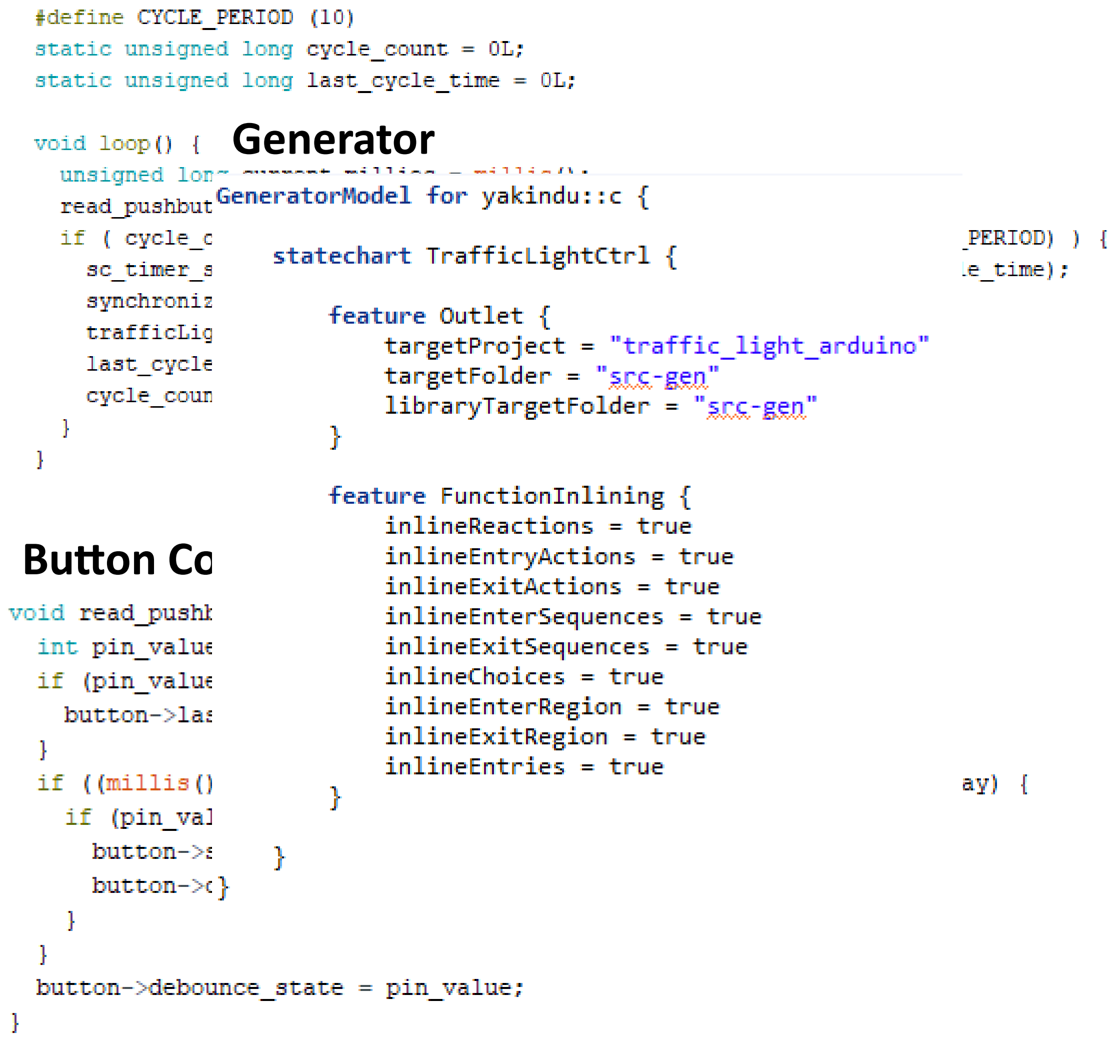
- pinMode*(pin_nr, mode)*
- digitalWrite(*pin_nr, {0, 1}*)
- digitalRead(*pin_nr*): {0, 1}





Deploying onto Hardware

Runner



```
#define CYCLE_PERIOD (10)
static unsigned long cycle_count = 0L;
static unsigned long last_cycle_time = 0L;

void loop() {
  unsigned lon
  read_pushbut
  if ( cycle_c                                                    PERIOD) ) {
    sc_timer_s                                                   e_time);
    synchroniz
    trafficLig
    last_cycle
    cycle_coun
  }
}
```

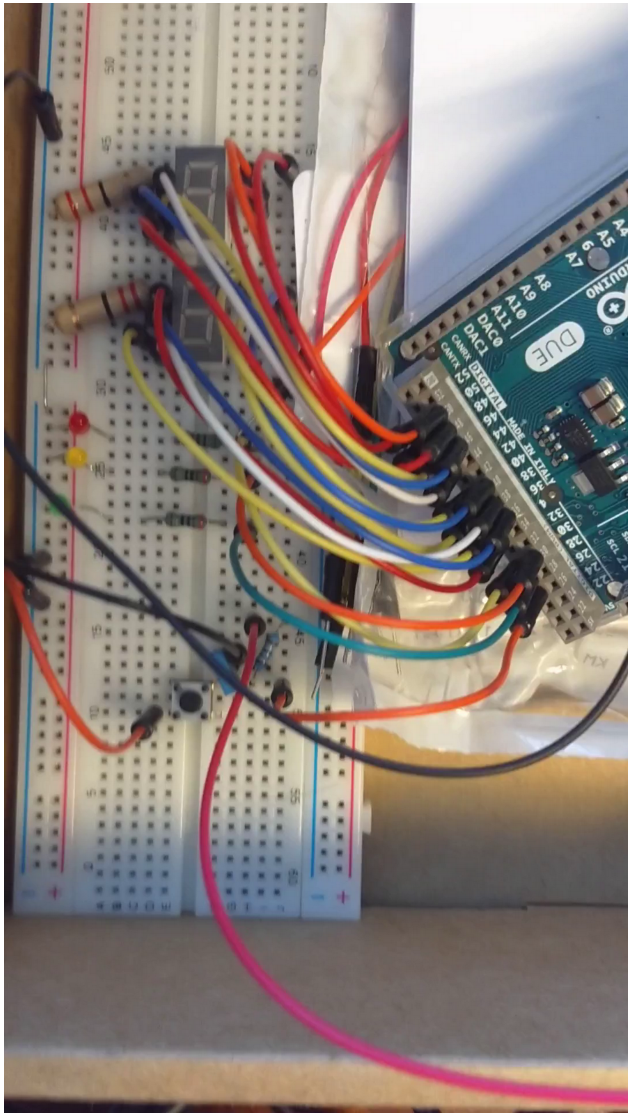
Generator

```
GeneratorModel for yakindu::c {

    statechart TrafficLightCtrl {

        feature Outlet {
            targetProject = "traffic_light_arduino"
            targetFolder = "src-gen"
            libraryTargetFolder = "src-gen"
        }

        feature FunctionInlining {
            inlineReactions = true
            inlineEntryActions = true
            inlineExitActions = true
            inlineEnterSequences = true
            inlineExitSequences = true
            inlineChoices = true
            inlineEnterRegion = true
            inlineExitRegion = true
            inlineEntries = true
        }
    }
}
```

Button Co

```
void read_pushb
  int pin_value
  if (pin_value
    button->las
  }
  if ((millis()                                                  ay) {
    if (pin_val
      button->s
      button->c
    }
  }
  button->debounce_state = pin_value;
}
```

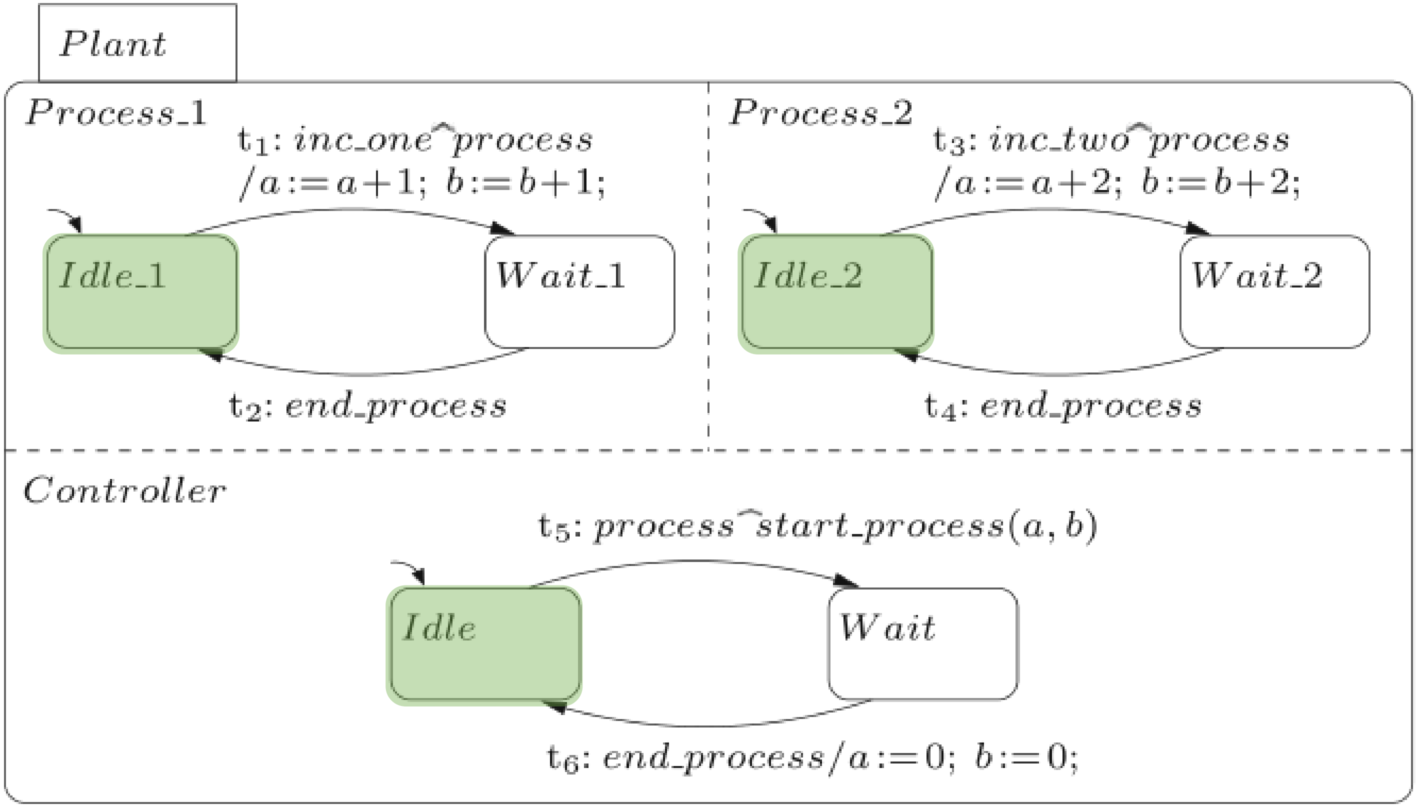
Deployed Application



Semantic Choices

Semantic Choices

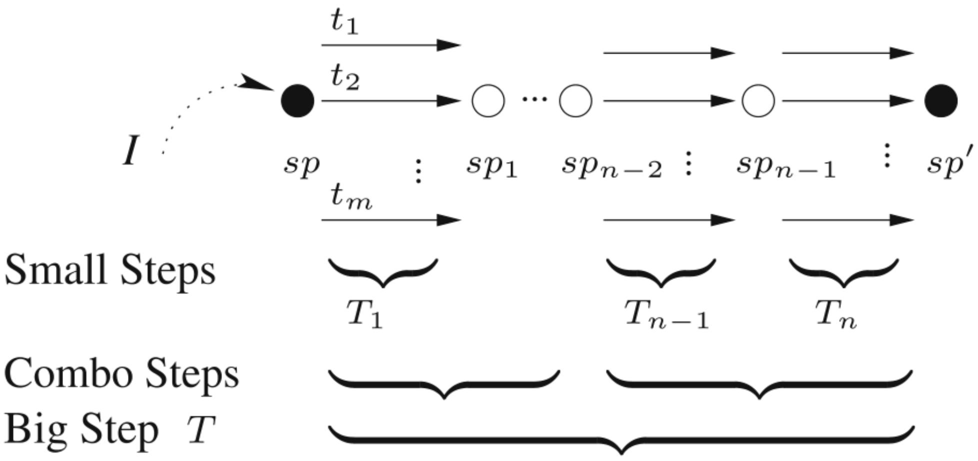
enabled events: [*inc_one*, *inc_two*]



Esmaeilsabzali, S., Day, N.A., Atlee, J.M. et al., Deconstructing the semantics of big-step modelling languages, Requirements Eng (2010) 15: 235.

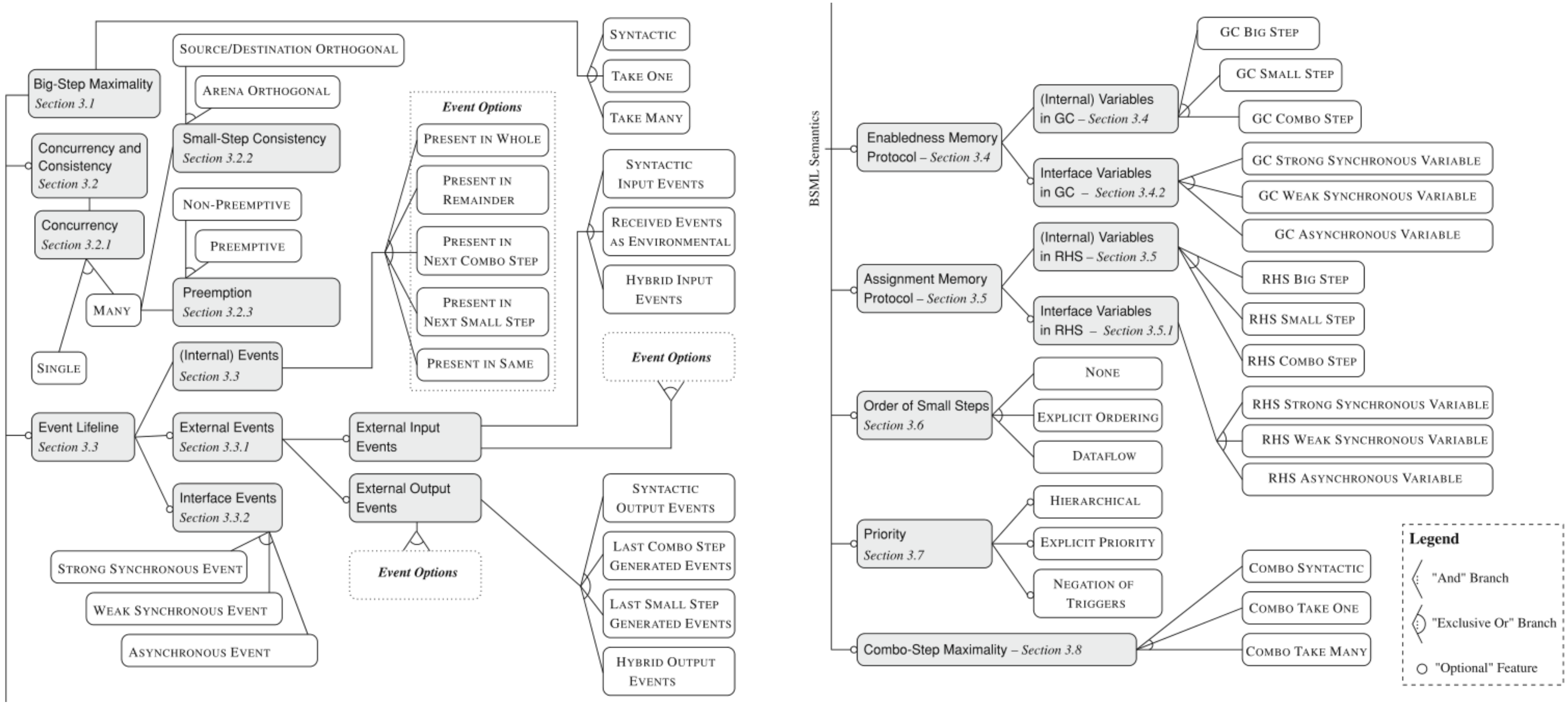
Big Step, Small Step

- A “big step” takes the system from one “quiescent state” to the next.
- A “small step” takes the system from one “snapshot” to the next (execution of a set of enabled transitions).
- A “combo step” groups multiple small steps.



Esmaeilsabzali, S., Day, N.A., Atlee, J.M. et al., Deconstructing the semantics of big-step modelling languages, Requirements Eng (2010) 15: 235.

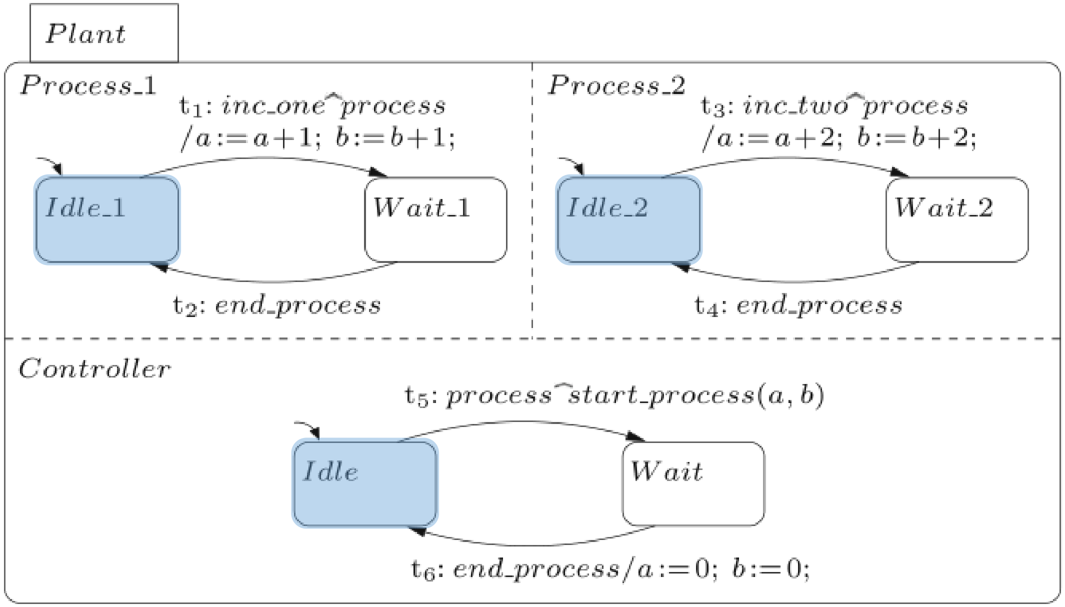
Semantic Options



Esmaeilsabzali, S., Day, N.A., Atlee, J.M. et al., Deconstructing the semantics of big-step modelling languages, Requirements Eng (2010) 15: 235.

Revisiting the Example

enabled events: [*inc_one*, *inc_two*]



concurrency: single

event lifeline: next combo step

assignment: RHS small step

-> <{t1}, {t3}, {t5}> and

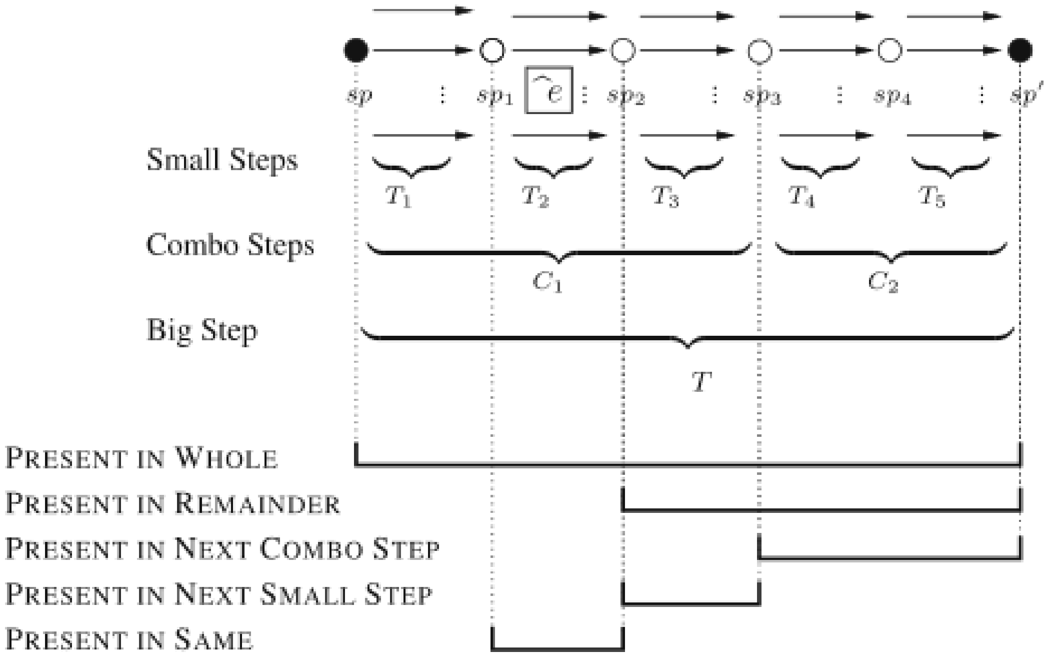
<{t3}, {t1}, {t5}>

event lifeline: present in remainder

-> <{t1}, {t5}, {t3}> becomes possible

Esmaeilsabzali, S., Day, N.A., Atlee, J.M. et al., Deconstructing the semantics of big-step modelling languages, Requirements Eng (2010) 15: 235.

Event Lifeline



Small Steps

Combo Steps

Big Step

PRESENT IN WHOLE

PRESENT IN REMAINDER

PRESENT IN NEXT COMBO STEP

PRESENT IN NEXT SMALL STEP

PRESENT IN SAME

Semantic Options: Examples

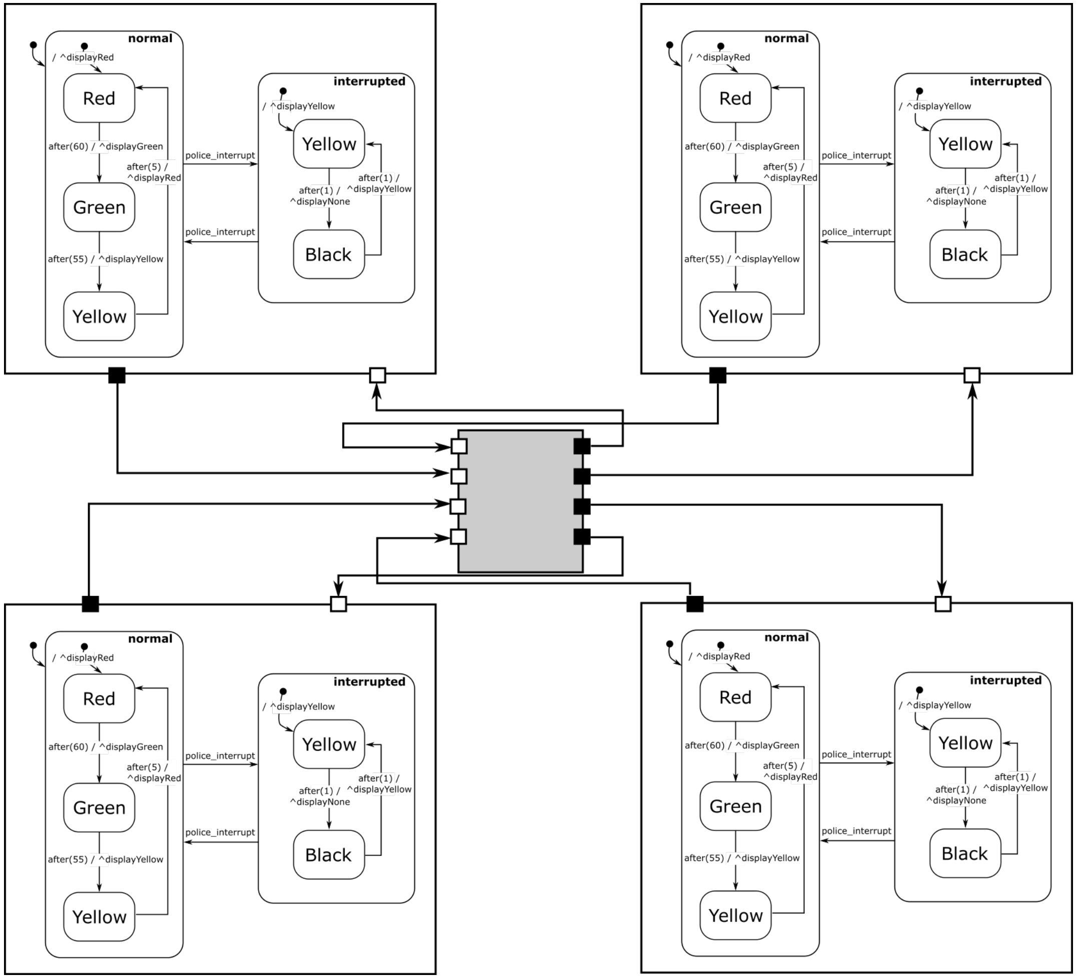
	Rhapsody	Statemate	(Default) SCCD
Big Step Maximality	Take Many	Take Many	Take Many
Internal Event Lifeline	Queue	Next Combo Step	Queue
Input Event Lifeline	First Combo Step	First Combo Step	First Combo Step
Priority	Source-Child	Source-Parent	Source-Parent
Concurrency	Single	Single	Single

Simon Van Mierlo, Yentl Van Tendeloo, Bart Meyers, Joeri Exelmans, and Hans Vangheluwe. SCCD: SCXML extended with class diagrams. In 3rd Workshop on Engineering Interactive Systems with SCXML, part of EICS 2016, 2016.

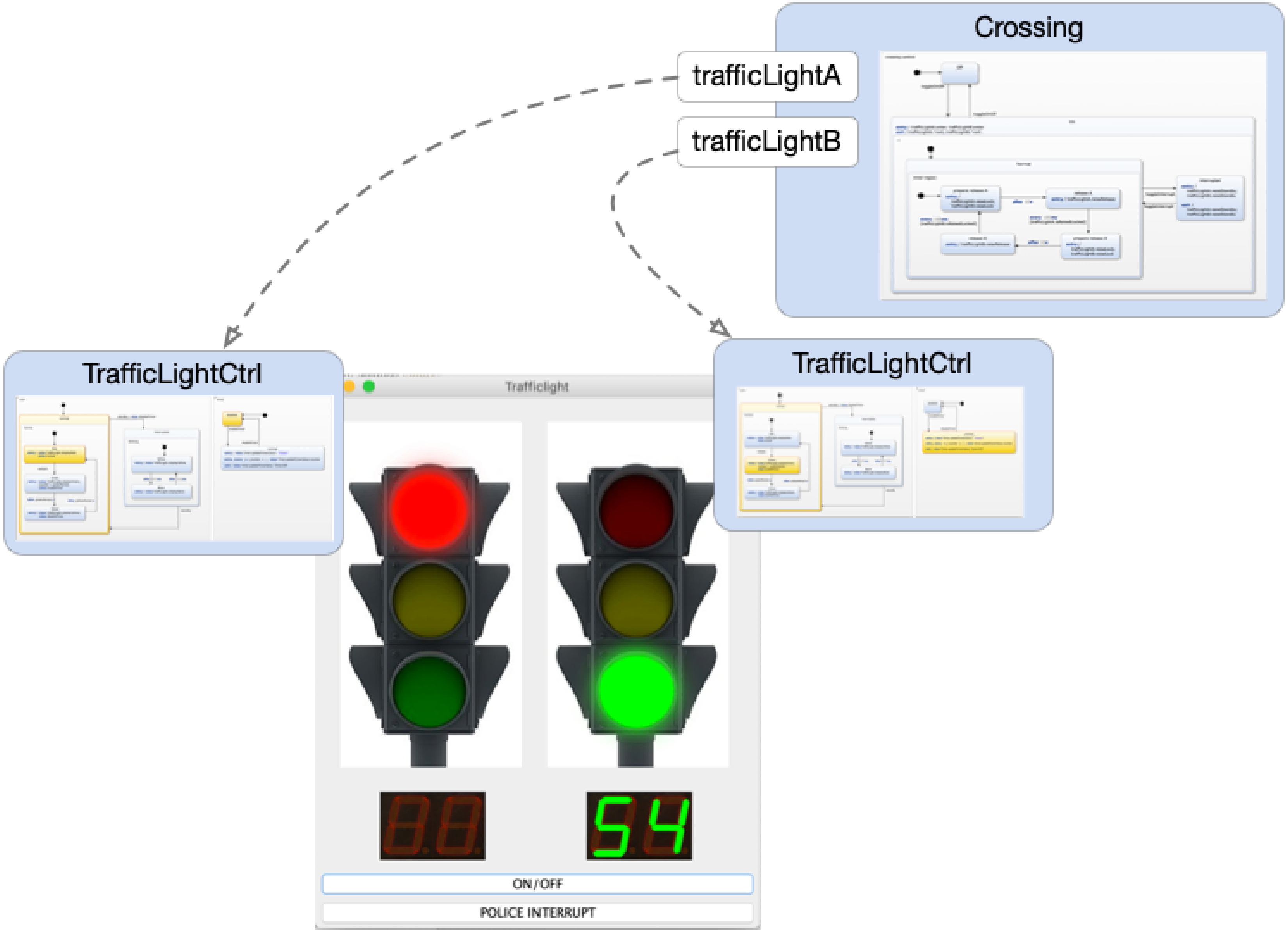
Composition

Composition of Statecharts

- Composition of multiple Statechart models
    - Instantiation
    - Communication
    - Semantics
- Often solved in code…



Composition Example



Crossing

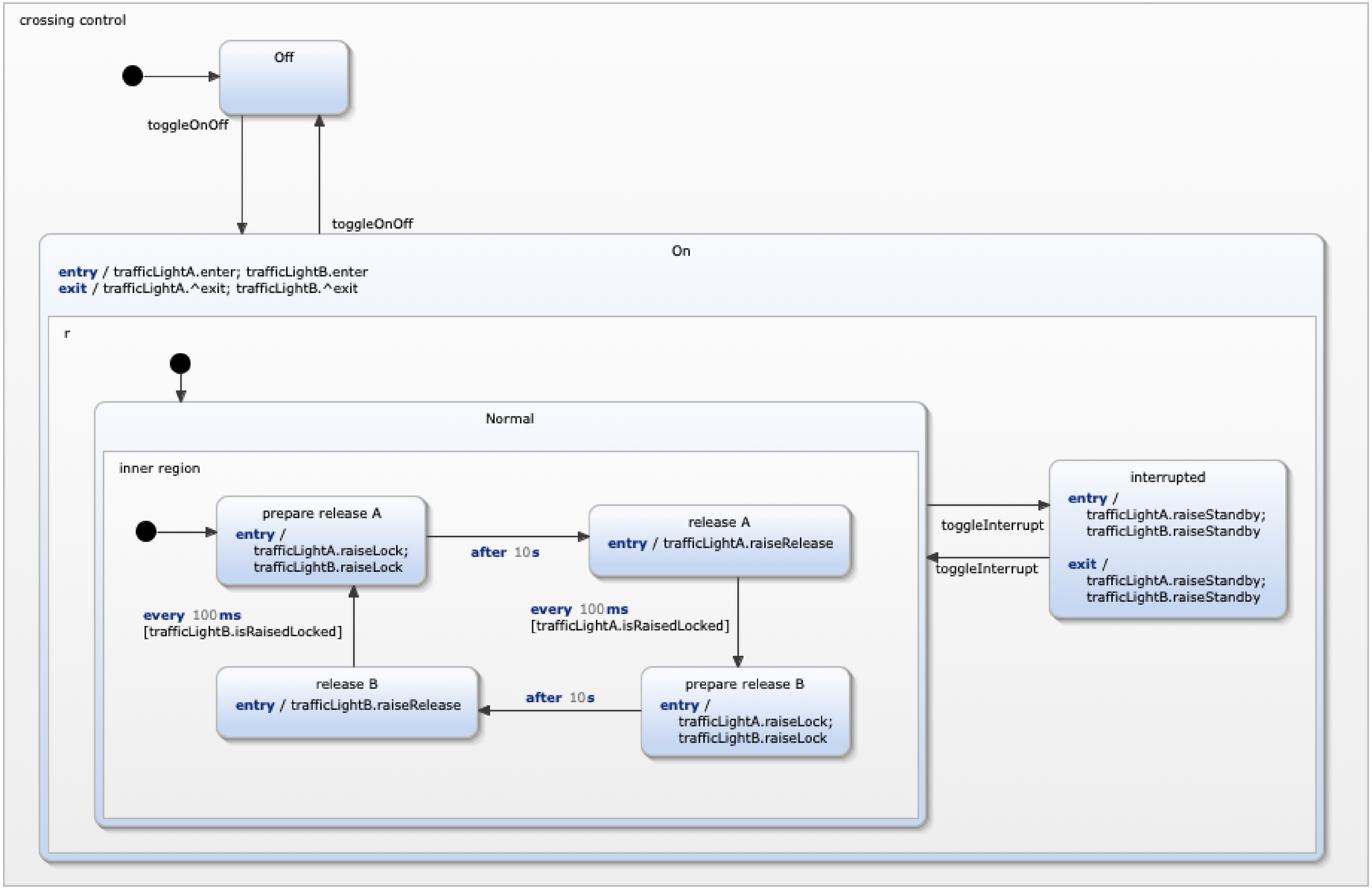
trafficLightA

trafficLightB

TrafficLightCtrl

TrafficLightCtrl

Composition Example



crossing control

Off

toggleOnOff

toggleOnOff

On

entry / trafficLightA.enter; trafficLightB.enter
exit / trafficLightA.^exit; trafficLightB.^exit

r

Normal

inner region

prepare release A
entry /
trafficLightA.raiseLock;
trafficLightB.raiseLock

after 10**s**

release A
entry / trafficLightA.raiseRelease

every 100**ms**
[trafficLightB.isRaisedLocked]

every 100**ms**
[trafficLightA.isRaisedLocked]

release B
entry / trafficLightB.raiseRelease

after 10**s**

prepare release B
entry /
trafficLightA.raiseLock;
trafficLightB.raiseLock

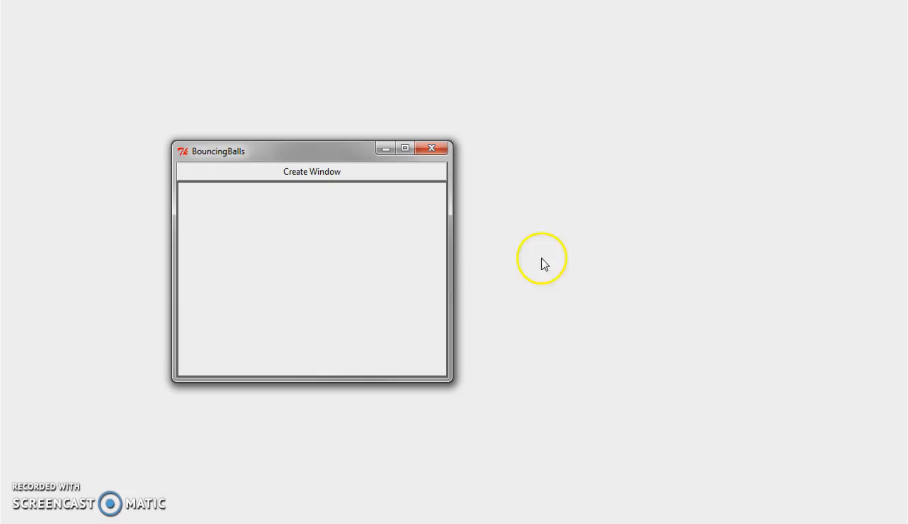
toggleInterrupt

toggleInterrupt

interrupted
entry /
trafficLightA.raiseStandby;
trafficLightB.raiseStandby

exit /
trafficLightA.raiseStandby;
trafficLightB.raiseStandby

Dynamic Structure: SCCD



Behavior

- Timed
- Autonomous
- Interactive
- Hierarchical

Structure

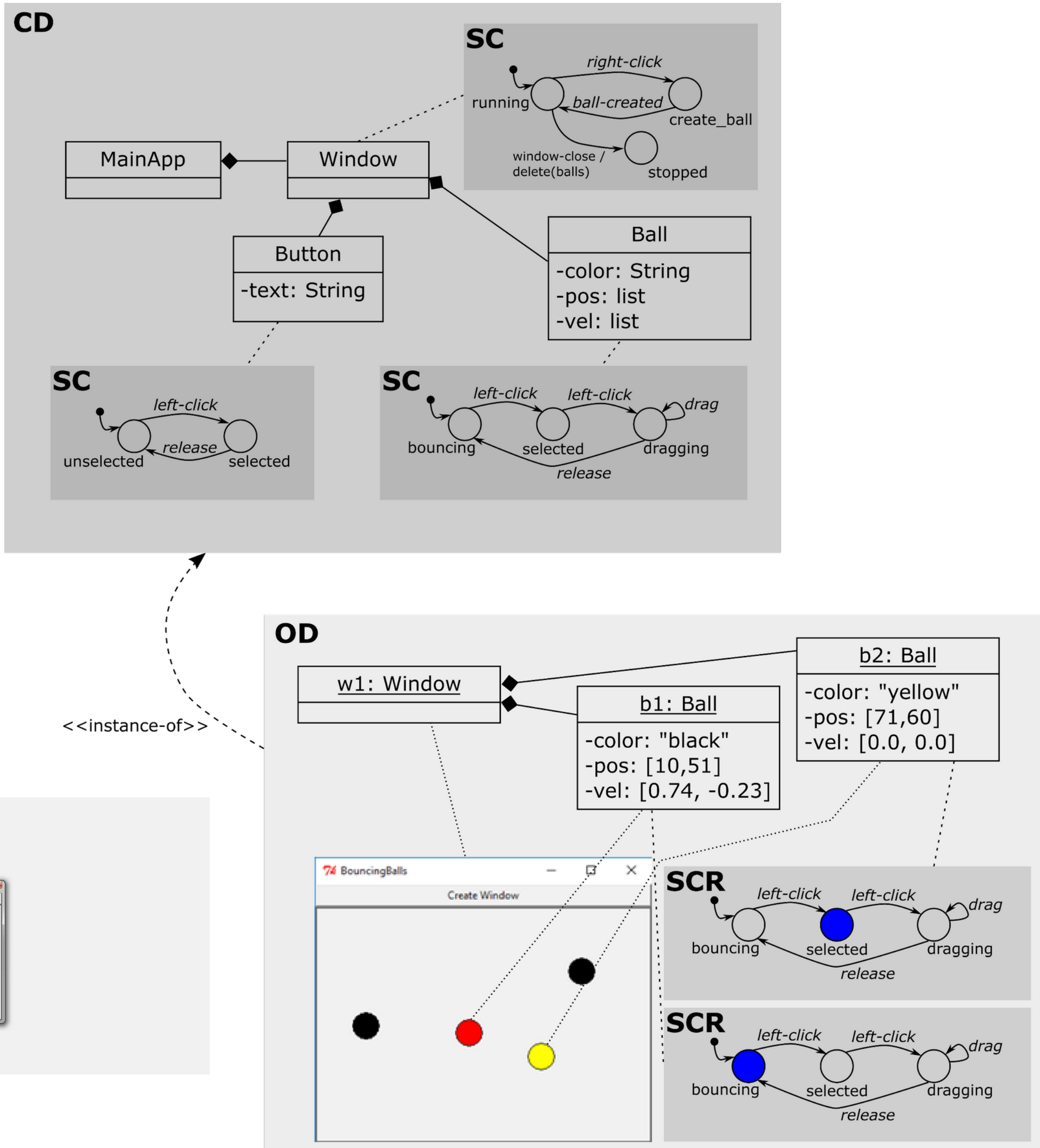
- Dynamic
- Hierarchical

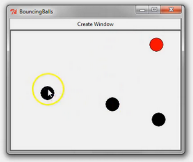
Design? Statecharts + ???

Coordination/Communication/Dynamic Structure often implemented in code...

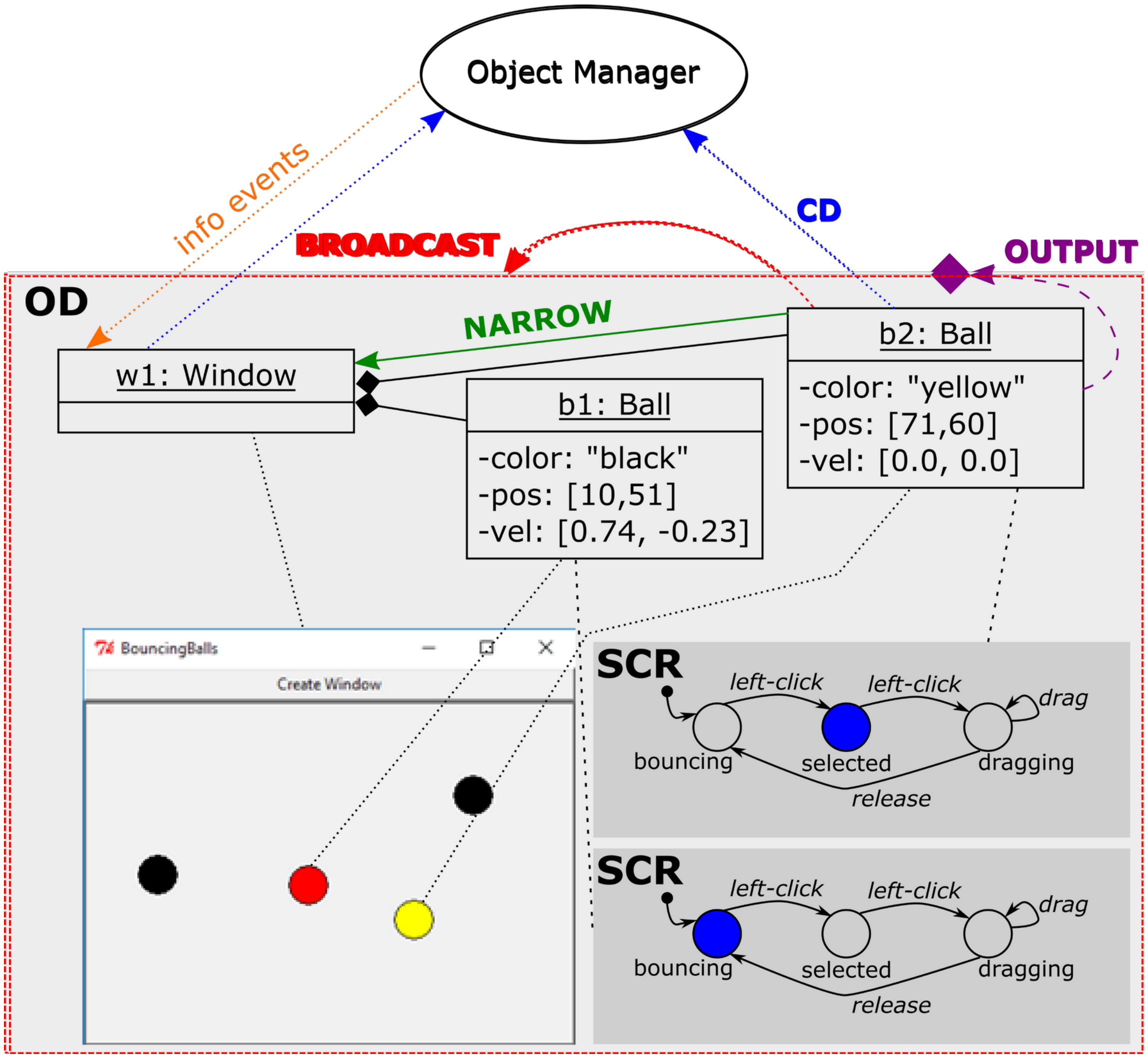
Simon Van Mierlo, Yentl Van Tendeloo, Bart Meyers, Joeri Exelmans, and Hans Vangheluwe. SCCD: SCXML extended with class diagrams. In 3rd Workshop on Engineering Interactive Systems with SCXML, part of EICS 2016, 2016.

SCCD: Conformance

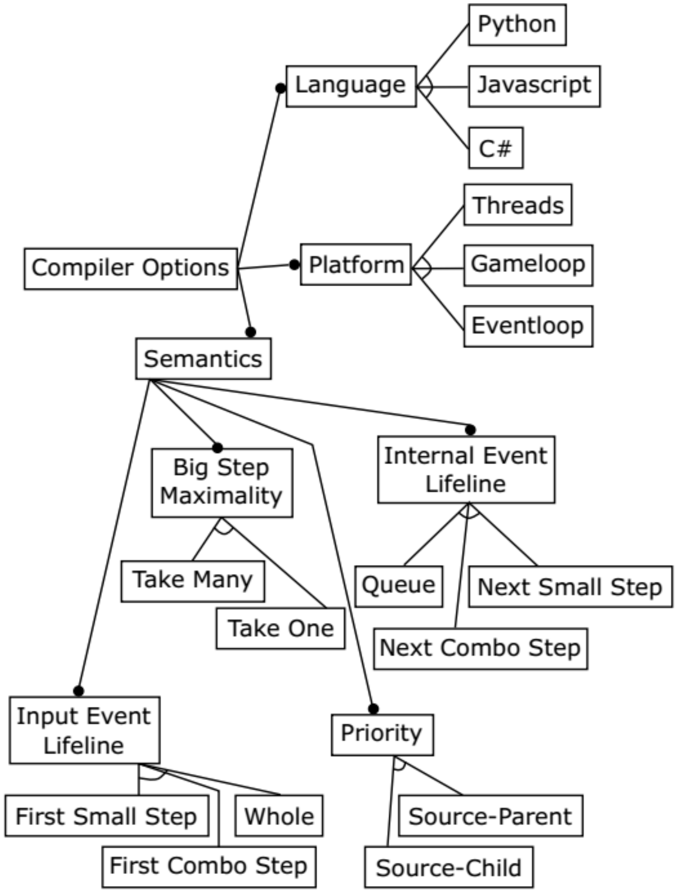


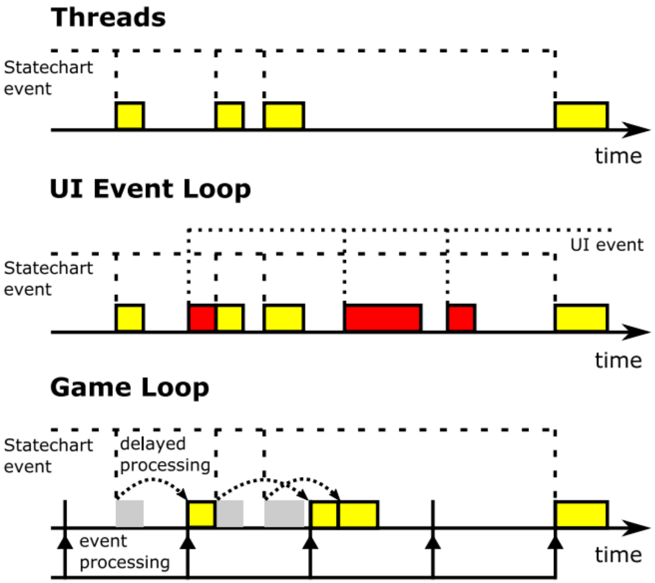


Communication: Event Scopes



SCCD Compiler

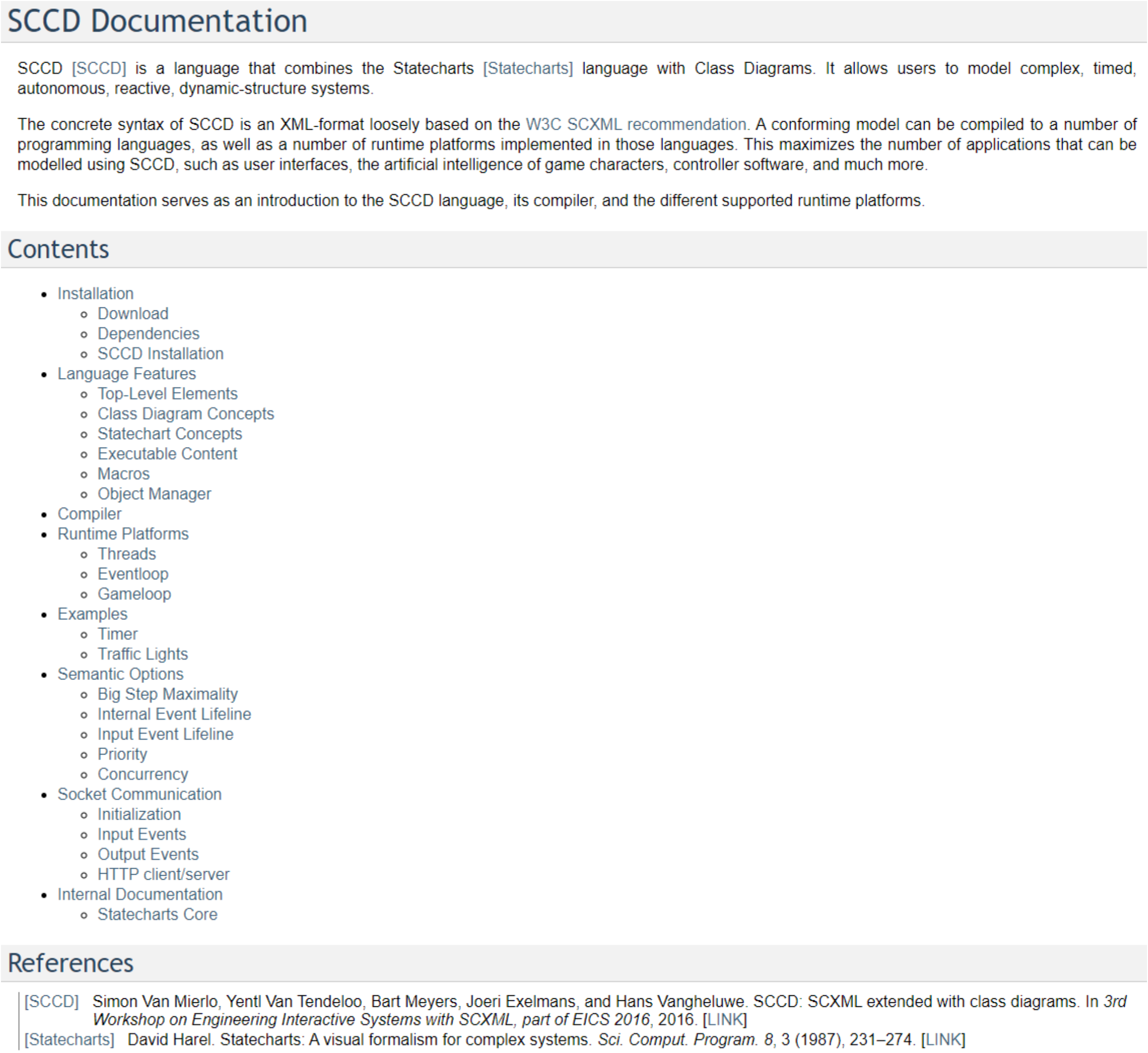




Simon Van Mierlo, Yentl Van Tendeloo, Bart Meyers, Joeri Exelmans, and Hans Vangheluwe. **SCCD: SCXML extended with class diagrams**. In *3rd Workshop on Engineering Interactive Systems with SCXML, part of EICS 2016*, 2016

SCCD

https://msdl.uantwerpen.be/documentation/SCCD/



SCCD Documentation

SCCD [SCCD] is a language that combines the Statecharts [Statecharts] language with Class Diagrams. It allows users to model complex, timed, autonomous, reactive, dynamic-structure systems.

The concrete syntax of SCCD is an XML-format loosely based on the W3C SCXML recommendation. A conforming model can be compiled to a number of programming languages, as well as a number of runtime platforms implemented in those languages. This maximizes the number of applications that can be modelled using SCCD, such as user interfaces, the artificial intelligence of game characters, controller software, and much more.

This documentation serves as an introduction to the SCCD language, its compiler, and the different supported runtime platforms.

Contents

References

[SCCD] Simon Van Mierlo, Yentl Van Tendeloo, Bart Meyers, Joeri Exelmans, and Hans Vangheluwe. SCCD: SCXML extended with class diagrams. In *3rd Workshop on Engineering Interactive Systems with SCXML, part of EICS 2016*, 2016. [LINK]

[Statecharts] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program. 8*, 3 (1987), 231–274. [LINK]

Recap

- Model the behaviour
  of complex, timed, reactive, autonomous systems
  - “What” instead of “How”
    (= implemented by Statecharts compiler)
- Abstractions:
  - States (composite, orthogonal)
  - Transitions
  - Timeouts
  - Events
- Tool support:
  - Yakindu
  - SCCD