Developing Reactive Systems Using Statecharts

Simon Van Mierlo
University of Antwerpc-
Flanders Make
Belgium
simon.vanmierlo@uantwerpen.be

Axel Terfloth
itemis AG
Germany
terfloth@itemis.de

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

16 September 2019

Universiteit Antwerpen | Ansymo Antwerp Systems & Software Modelling University of Antwerp | Nexor Cyber-Physical Systems University of Antwerp | FLANDERS MAKE MANUFACTURING INNOVATION NETWORK | McGill | itemis

Time Table

09:00 – 10:30	Introduction, Yakindu set-up
10:30 – 11:00	Coffee Break
11:00 – 12:30	Tutorial: Statecharts Concepts + Exercises
12:30 – 14:00	Lunch
14:00 – 15:30	Tutorial: Statecharts Concepts + Exercises
15:30 – 16:00	Coffee Break
16:00 – 17:30	Tutorial: Advanced Concepts

Introduction

Reactive Systems

- Com
- 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 timeouts (OS)?

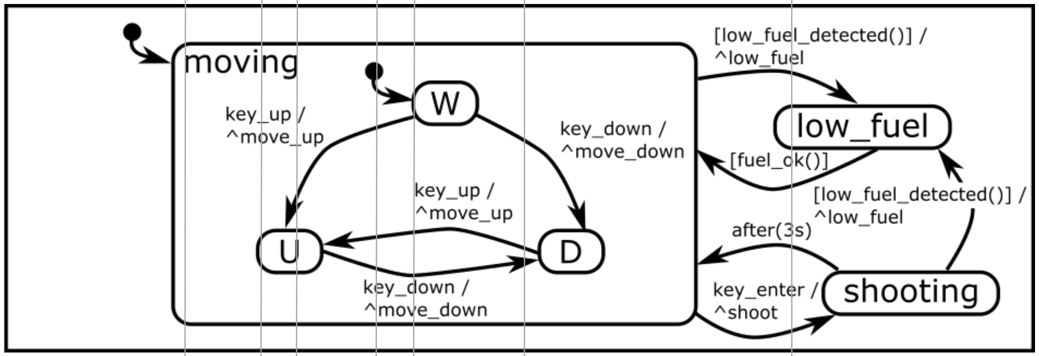
Programming language (and OS) is too low-level
-> most appropriate formalism: "what" vs. "how"

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

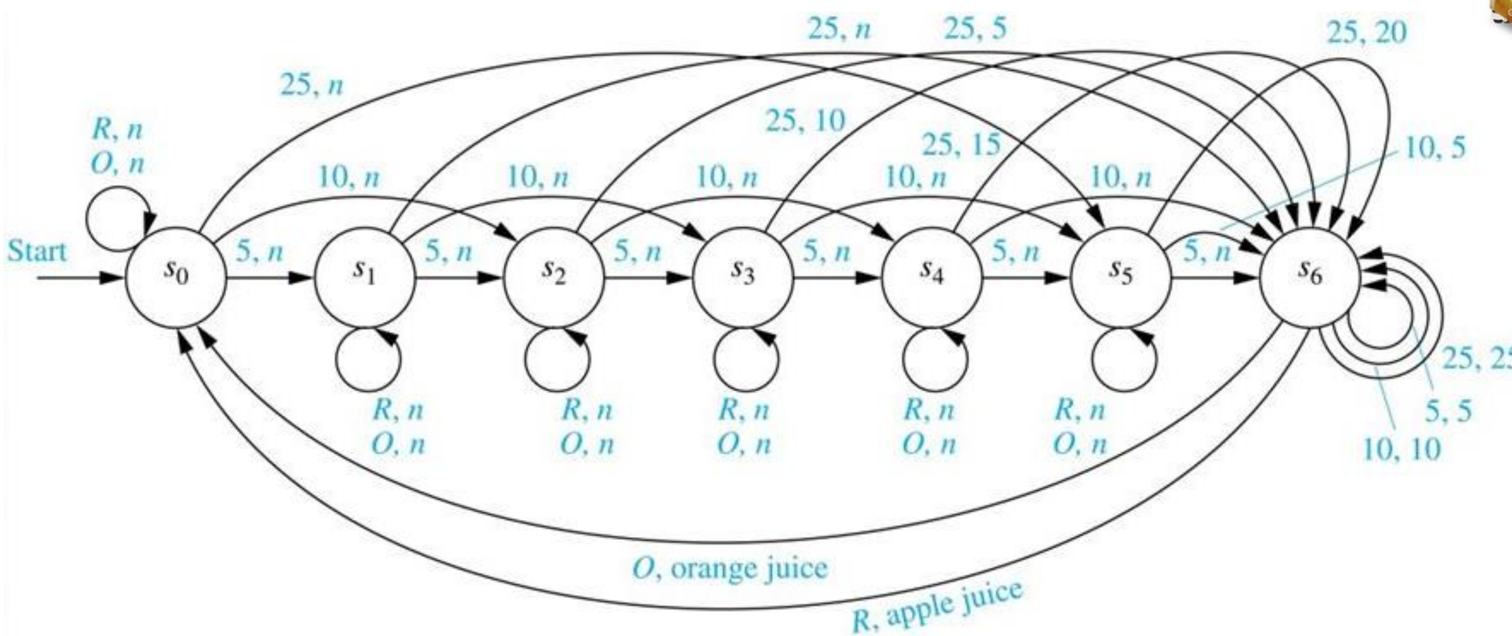
E. A. Lee, "The problem with threads," in *Computer*, vol. 39, no. 5, pp. 33-42, May 2006.

Discrete-Event Abstraction

behavioural model

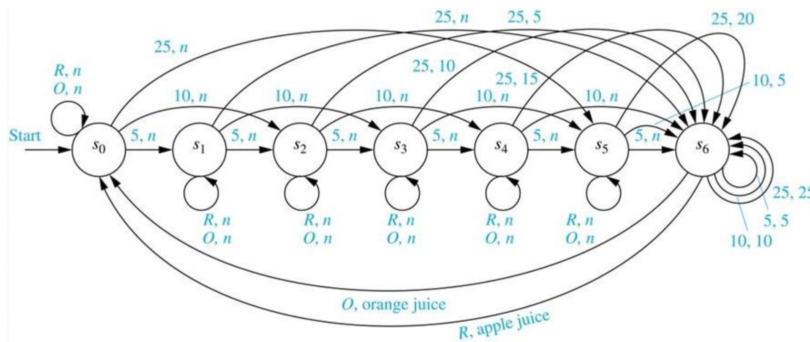


State Diagrams



- 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)

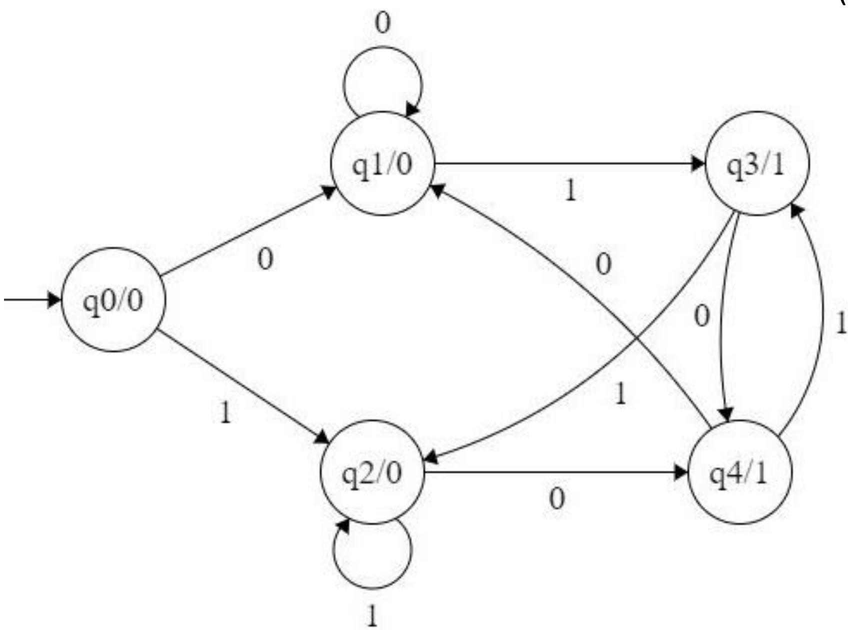
Alternative Representation: Parnas Tables

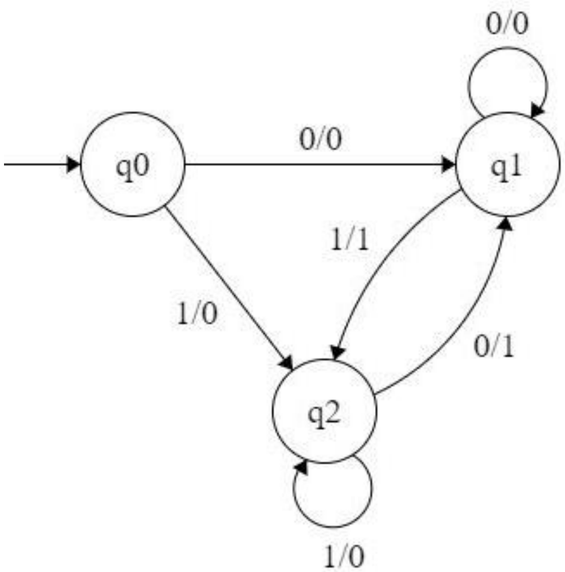


event/state	s_0	s_1	s_2	s_3	s_4	s_5	s_6
5	s_1, n	s_2, n	s_3, n	s_4, n	s_5, n	s_6, n	s_6, 5
10	s_2, n	s_3, n	s_4, n	s_5, n	s_6, n	s_6, 5	s_6, 10
25	s_5, n	s_6, n	s_6, 5	s_6, 10	S_6, 15	s_6, 20	s_6, 25
O	s_0, n	s_1, n	s_2, n	s_3, n	s_4, n	s_5, n	s_0, orange juice
R	s_0, n	s_1, n	s_2, n	s_3, n	s_4, n	s_5,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

Mealy and Moore Machines

FSA: (Q, q0, ∑, O, δ, λ)





Moore Machines

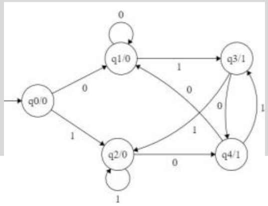
- Output only depends on current state. λ: Q → O
- Input: 00 -> Output: 111

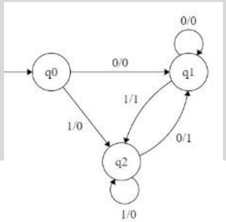
Mealy Machines

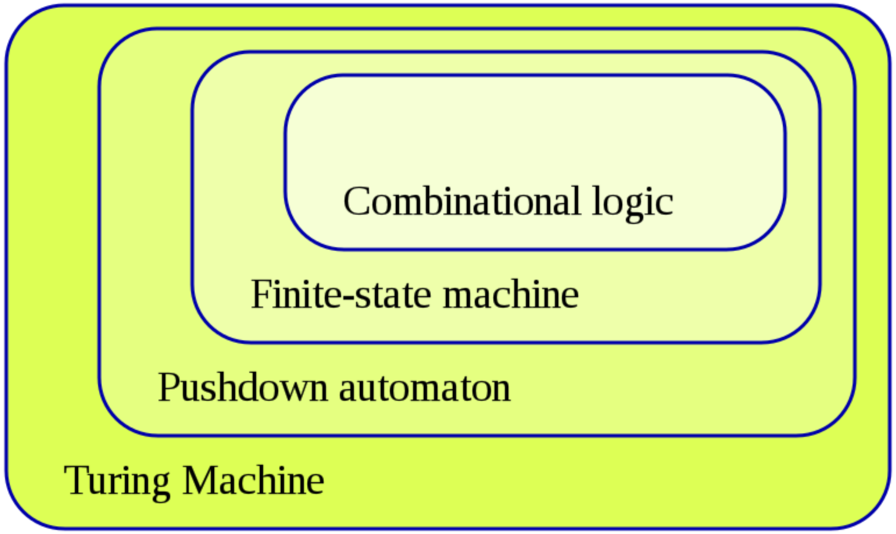
- Output depends current state and current input. λ: Q x ∑ → O
- Input: 00 -> Output: 11

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

FSAs: Expressiveness





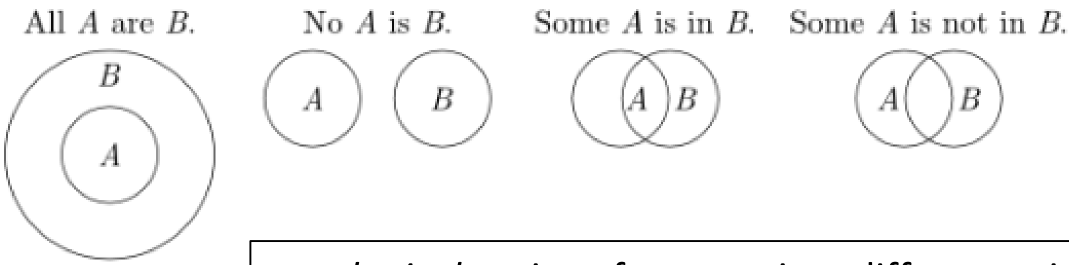


- Statecharts can be made turing-complete
  -> data memory, control flow, branching
- Extends FSAs
  -> borrows semantics from Mealy and Moore machines

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

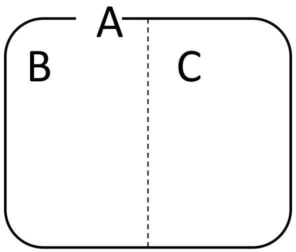
Higraphs

Euler Diagrams



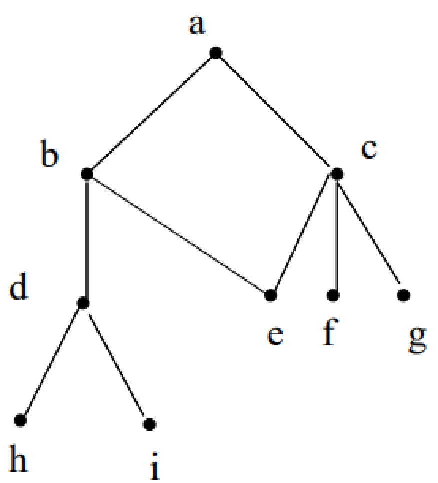
topological notions for set union, difference, intersection

Unordered Cartesian Product

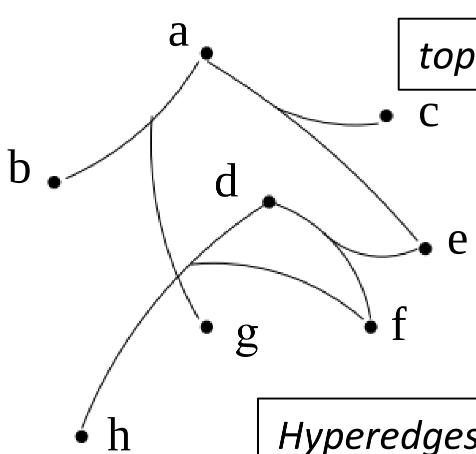


A = B x C

Hypergraphs



a graph



a hypergraph

topological notion (syntax): connectedness

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

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

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

Higraphs

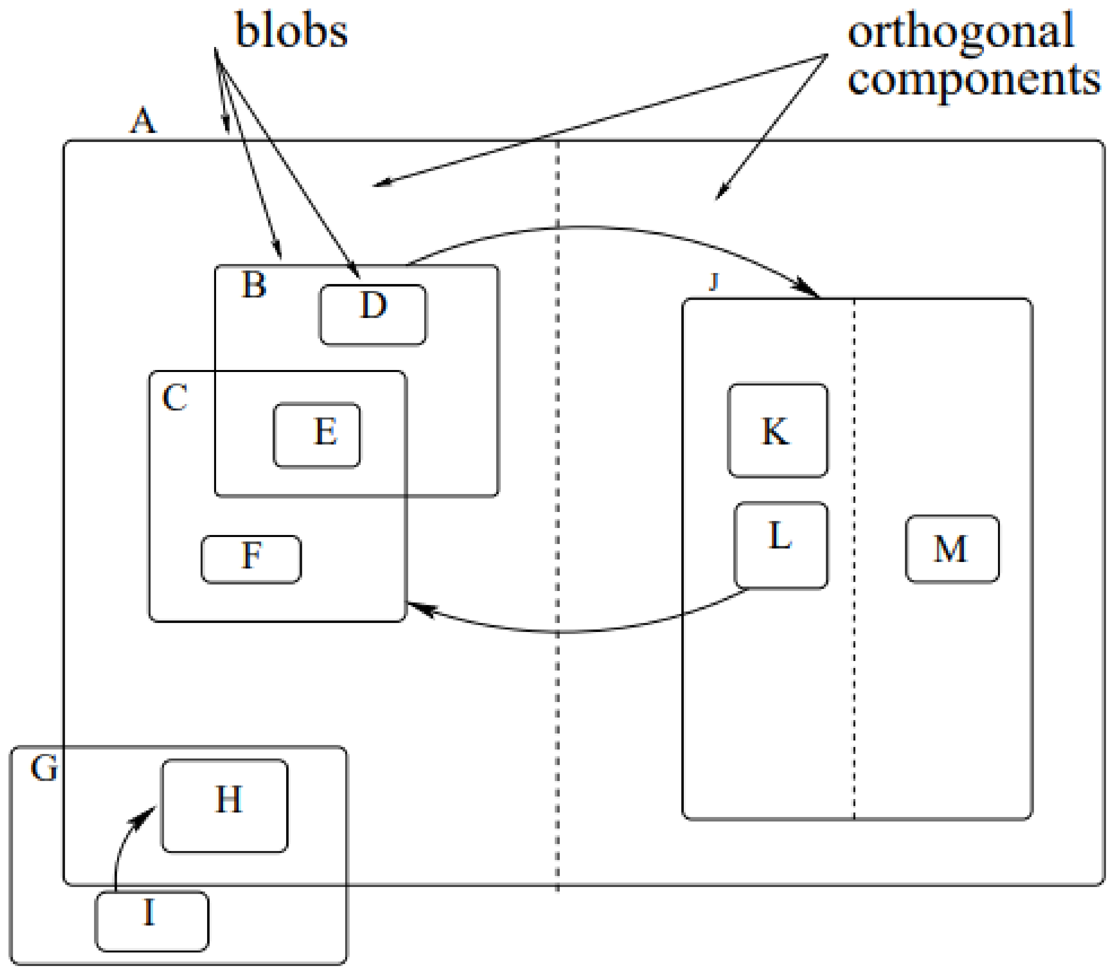
Euler Diagrams

+

Hypergraphs

+

Unordered Cartesian Product

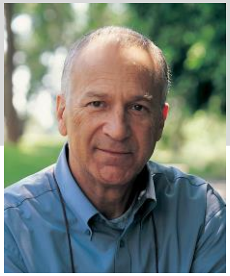


David Harel. On Visual Formalisms. Communications of the ACM. Volume 31, No. 5. 1988. pp. 514 - 530.

Statecharts

- Visual (topological, not geometric) formalism
- Precisely defined syntax and semantics
- Many uses:
  - Documentation (for human communication)
  - ~~Analysis (of behavioural properties)~~
  - 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 extend 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/

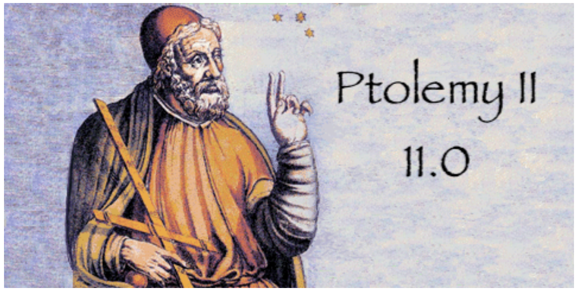
Statechart (Variants) Tools

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
ml



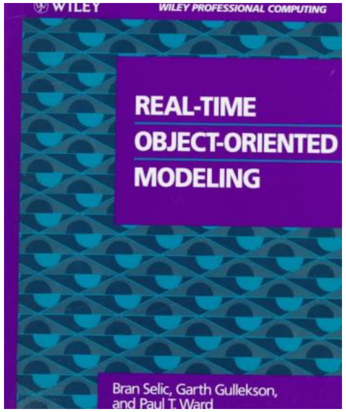
https://ptolemy.berkeley.edu/ptolemyII/ptII11.0/index.htm

YAKINDU STATECHART TOOLS

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

PAPYRUS REALTIME

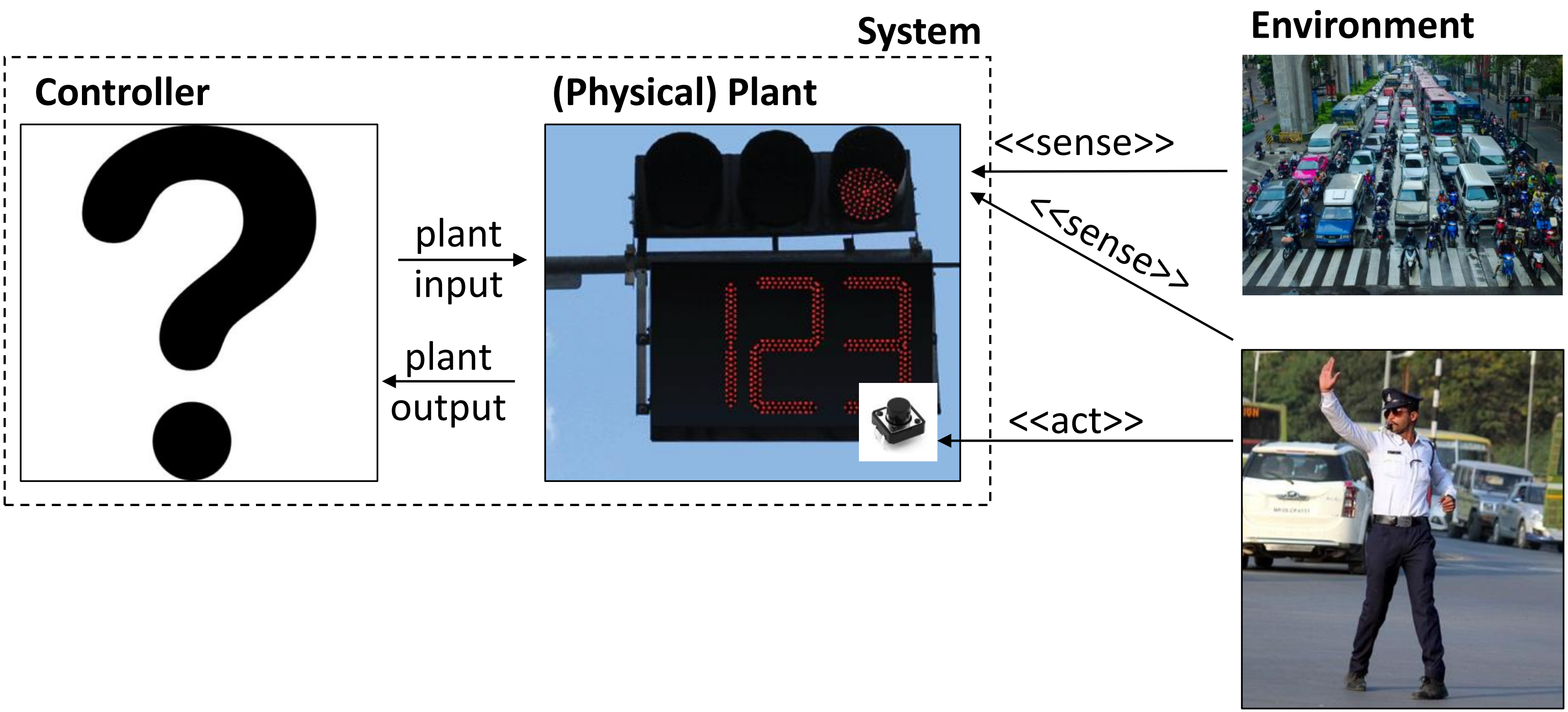
https://www.eclipse.org/papyrus-rt/



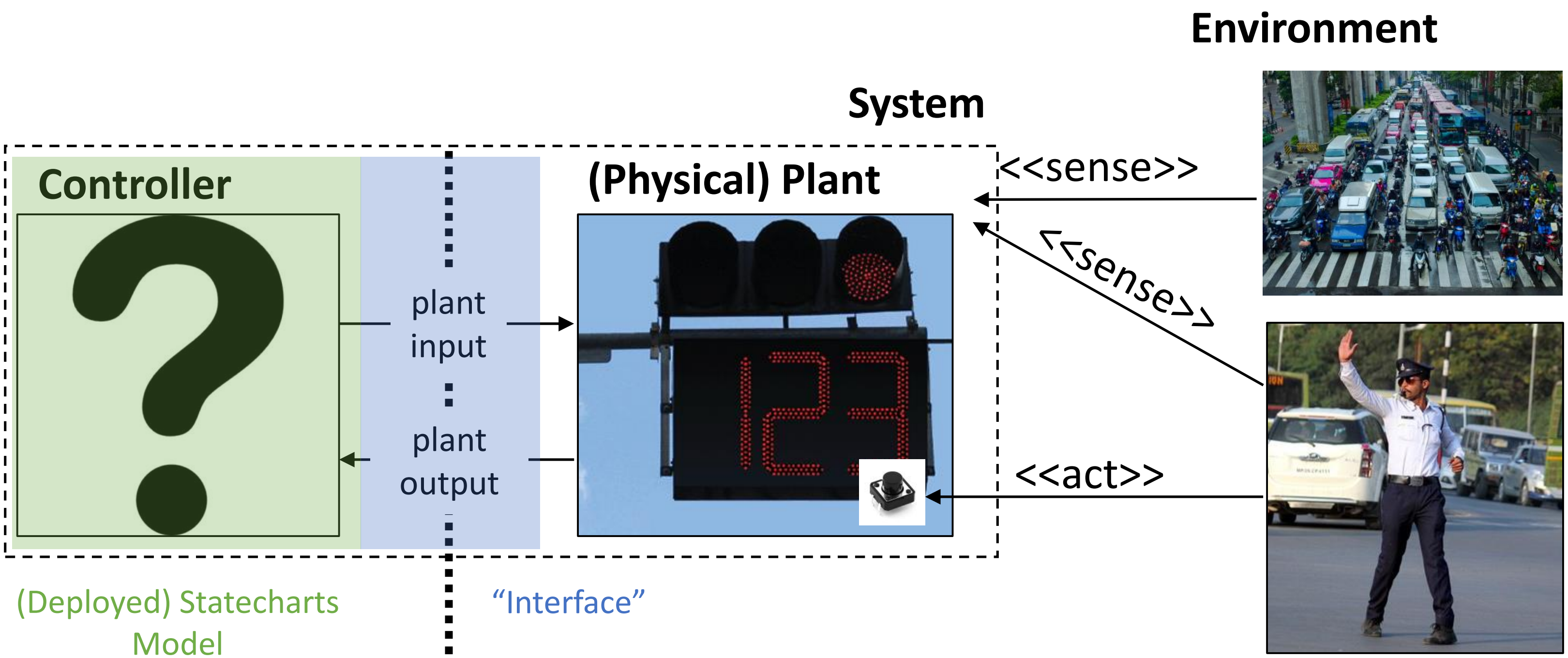
etrice.

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

Running Example



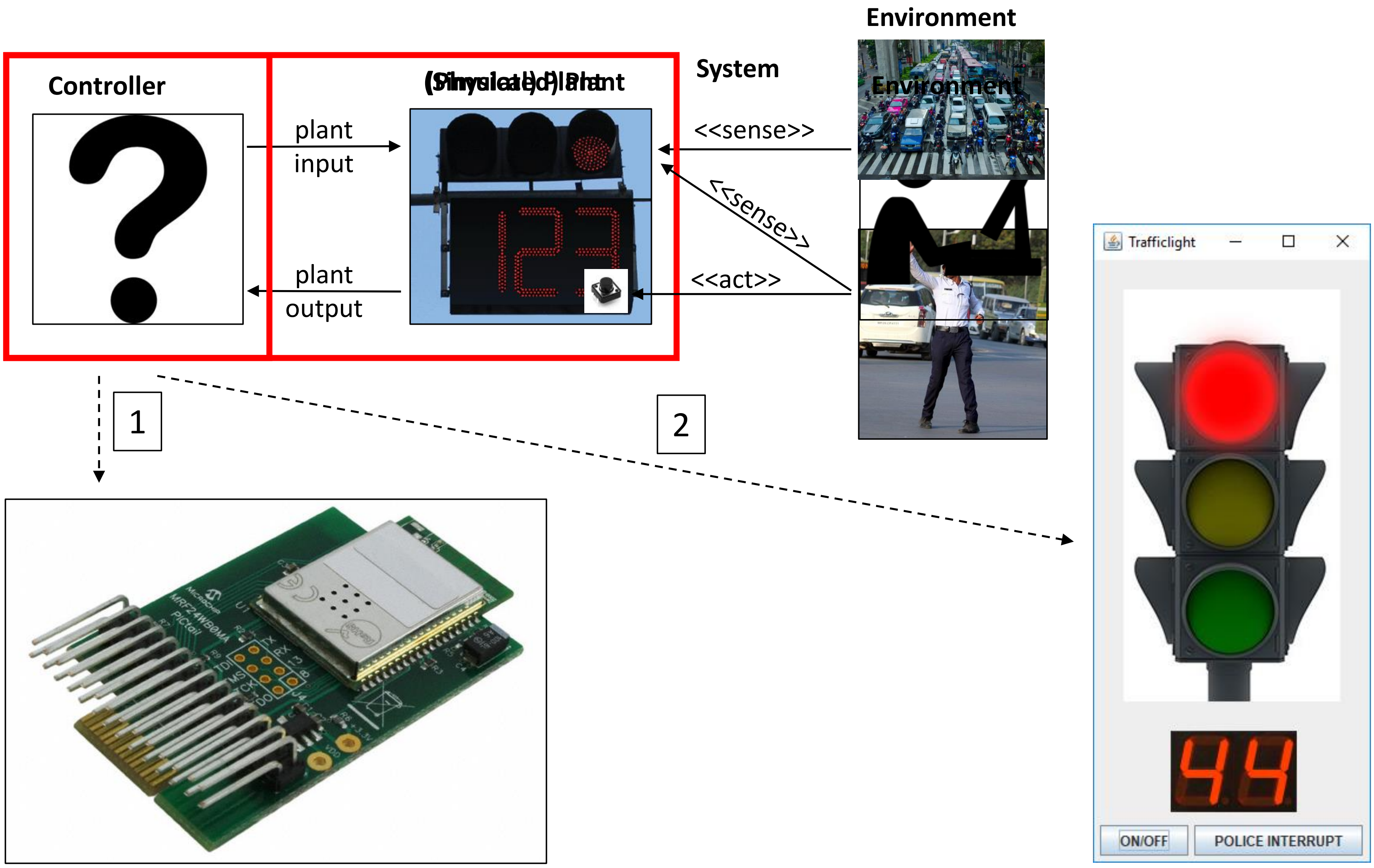
What are we developing?



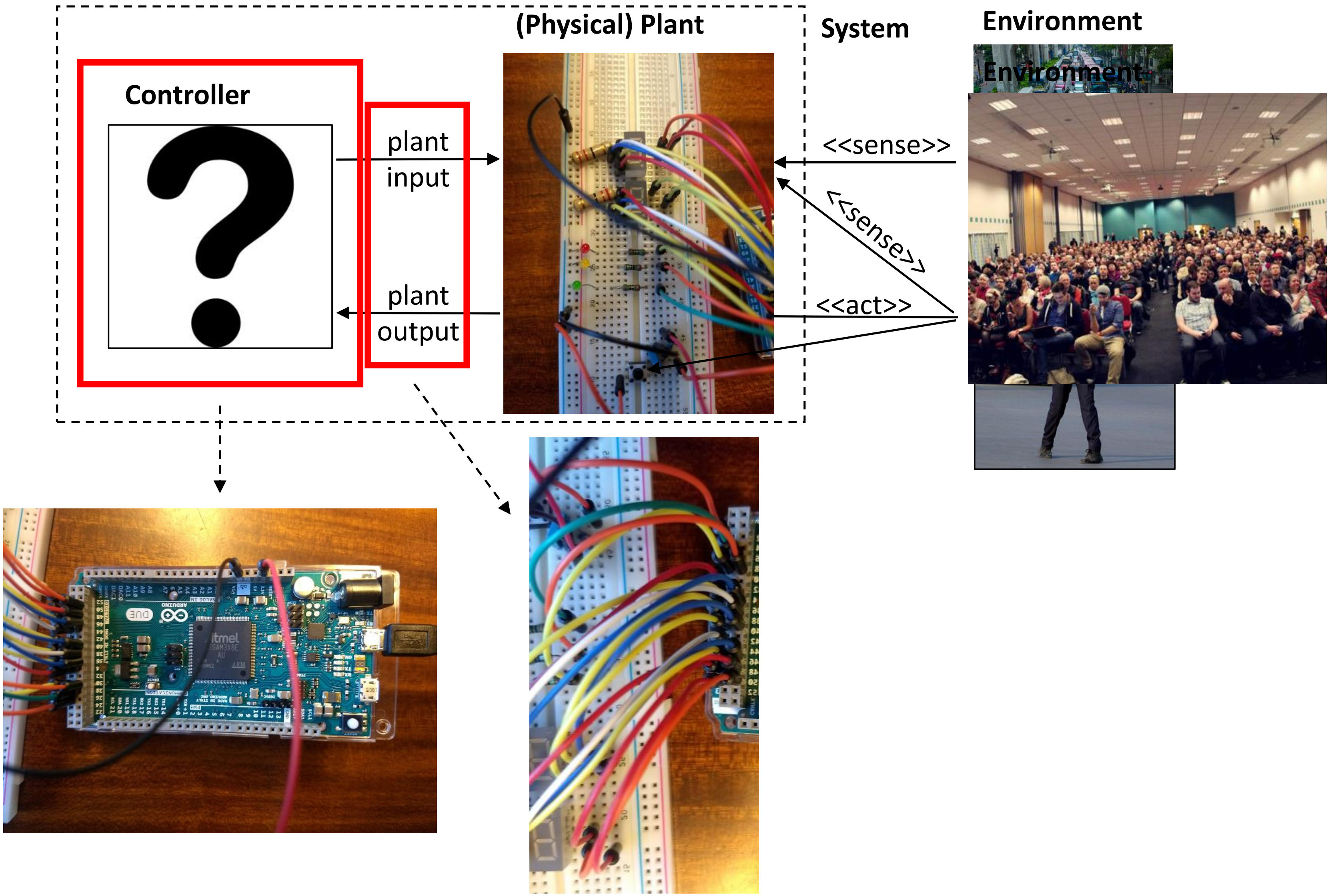
- Autonomous (timed) behaviour
- Interrupt logic
- Orthogonal (traffic light/timer) behaviour

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

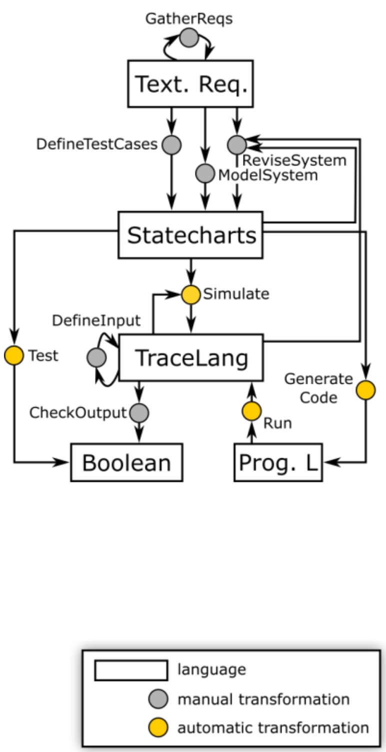
Deployment (Simulation)

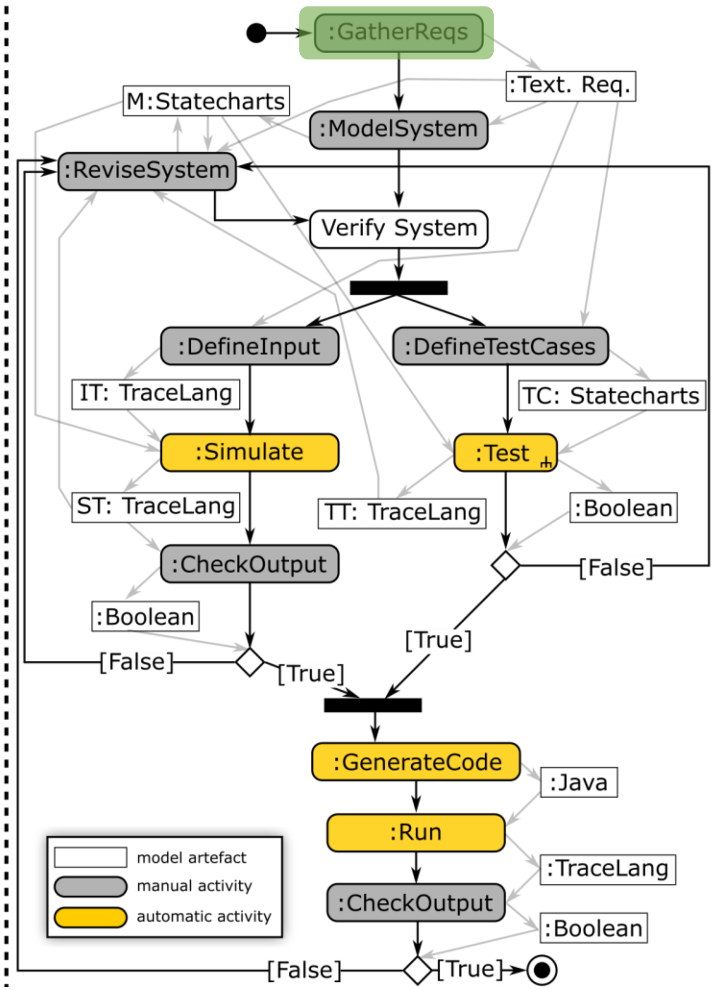


Deployment (Hardware)



Workflow

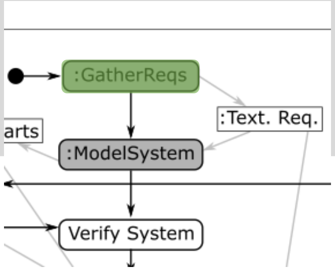


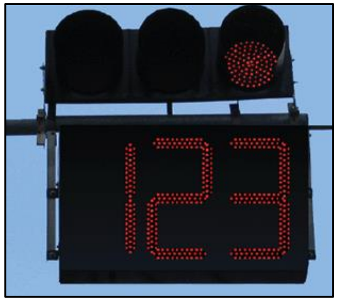


Hans Vangheluwe and Ghislain C. Vansteenkiste. A multi-paradigm modeling and simulation methodology: Formalisms and languages. In European Simulation Symposium (ESS), pages 168-172. Society for Computer Simulation International (SCS), October 1996. Genoa, Italy.

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



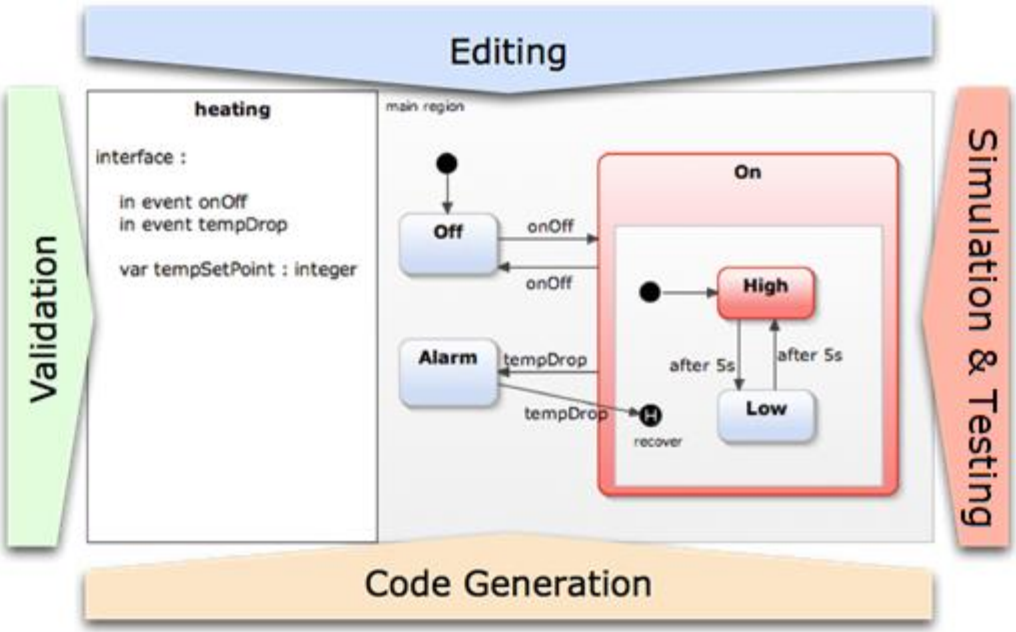


- R1: three differently coloured lights: red, green, yellow
- R2: at most one light is on at any point in time
- R3: at system start-up, the red light is on
- R4: cycles through red on, green on, and yellow on
- R5: red is on for 60s, green is on for 55s, yellow is on for 5s
- R6: time periods of different 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...

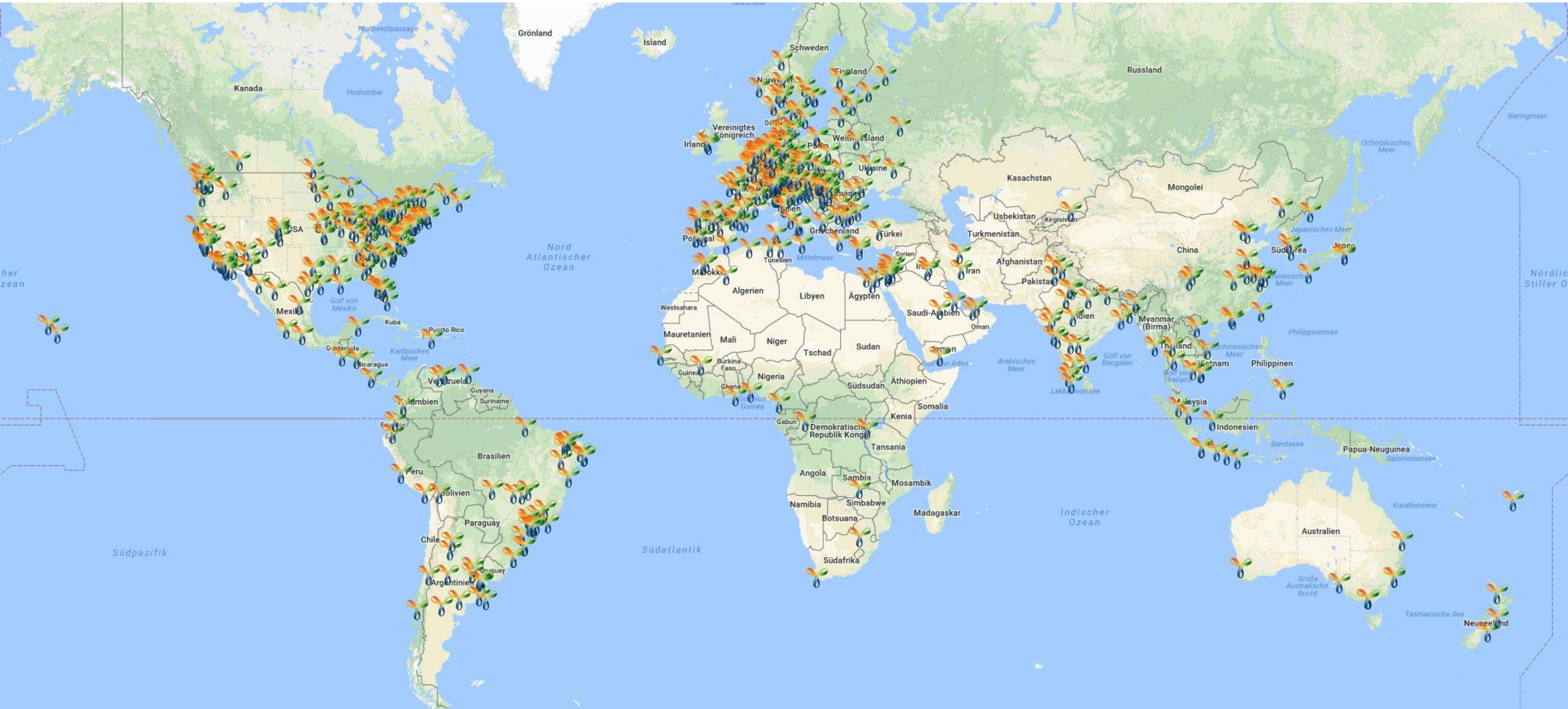
YAKINDU Statechart Tools provides an **integrated modeling environment** for the specification and development of **reactive, event-driven systems** based on the concept of statecharts.



Users from Trondheim to Christchurch and from Hawaii to Fiji

Customers from different domains:

Automotive, Avionic, Medical, Automation, Academia …



Academia

More than 300 universities are using YAKINDU Statechart Tools in research and education

" I like YAKINDU Statechart Tools a lot, especially the fact that it is simple and direct, and it is not burdened with some of the more advanced, expensive and heavy features of other professional statechart tools. "

Professor David Harel – inventor of statecharts
Vice president Israel Academy of Sciences and Humanities

•Active cooperation in research projects for predevelopment of future features like

- Model checking
- Scenario-based modeling
- Variability
- …

Industry Users

"After integrating YAKINDU Statechart Tools we could offer the possibility to develop state machines by use of a graphical notation – this makes it a lot easier for our users to master complexity."

Abhik Dey
Product Owner ASCET Developer at ETAS GmbH

Continental The Future in Motion

HARMAN A SAMSUNG COMPANY

Dräger

GIRA

ETAS

socomec Innovative Power Solutions

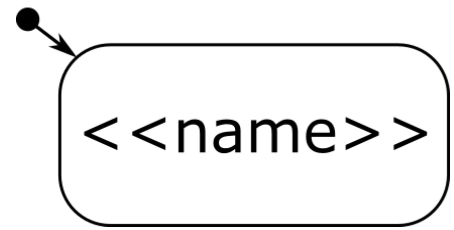
KOSTAL

KSB

Serenergy®

The Statecharts Language

States



being **in** a state

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

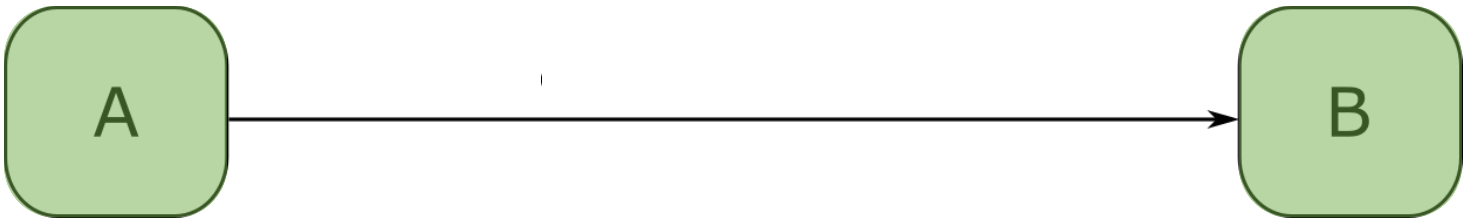
= the system is in **configuration** *<<name>>*

initial state

exactly one per model

“entry point”

Transitions

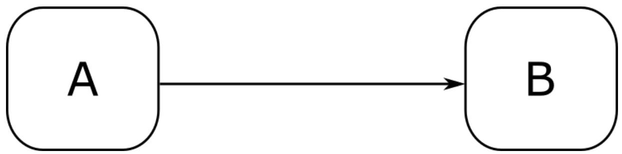


- Model the **dynamics** of the system:
  - *if*
    - the system is **in state A**
    - and **event** is **processed**
  - *then*
    1. **output_action** is evaluated
    2. and the new **active state** is B

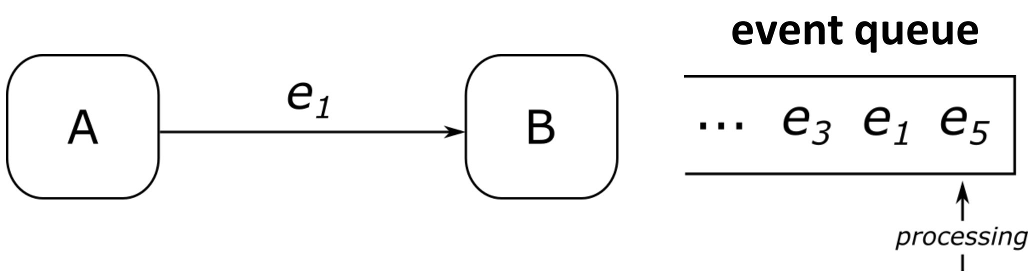
Transitions: Events

event(params) / output_action(params)

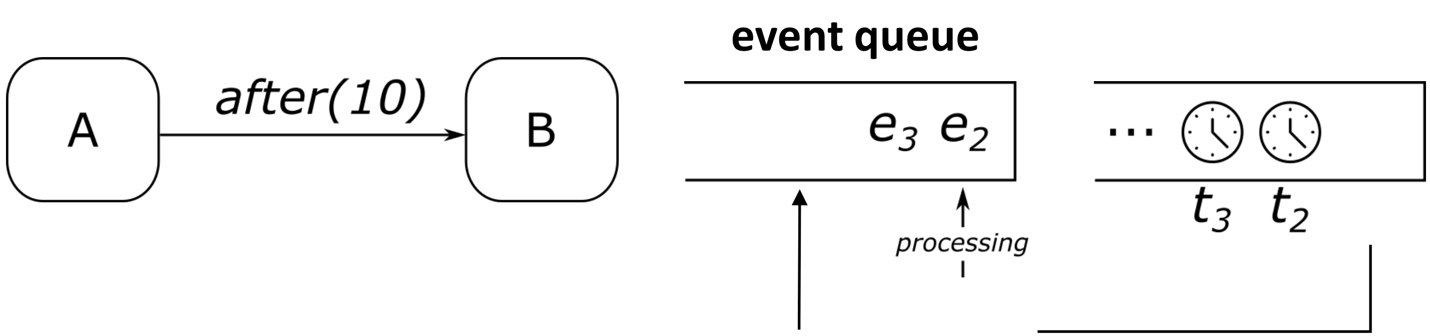
- Spontaneous



- Input Event

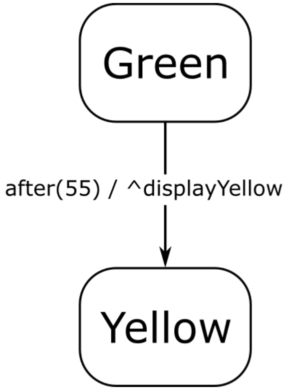


- After Event



Transitions: Raising Output Events

event(params) / output_action(params)



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

The YAKINDU Statechart Dialect

Reaction Syntax

In YAKINDU transitions are reactions.

- reactions define an effect that is executed when a specified trigger occurs and/or a guard condition becomes true
- a transitions effect always includes transition from the source state to the target state.

The reaction syntax is: *trigger* [*guard*] / *effect*

- trigger: list of events
- guard: boolean expression (explained later)
- effect: some action that produce an effect (incl. state transition
- … all optional.

Transition Trigger

trigger [*guard*] / *effect*

The following trigger types exist for transitions:

```
buttonPressed     // named event triggers

after 2s          // one shot time trigger
every x ms        // periodic time trigger

always            // pseudo trigger that is always active
oncycle           // same as ‚always‘

else              // pseudo trigger for choices
default           // same as ‚else‘
```

Trigger can be a list of events:

buttonPressed, systemAlert, after 20 s <=> buttonPressed OR systemAlert OR after 20 s *trigger [guard] / effect*

An transition effect includes:

1. The state transition
2. Zero or more actions (';' separated)

Actions can be:

```
raise event1   // raise events
x = 100   // variable assignment
doX(x, 10)   // operation call
```

Declaring Named Events

YAKINDU SCT requires the declaration of events

- events have visibility
  - public : defined on interface
  - private : defined internally
- public events have a direction
  - in
  - out
- events may have payload

```
// an incoming event
in event button_pressed
// an incoming event with payload
in event temperature_change : integer
// an outgoing event
out event halt_system
```

Prepare Exercises

YAKINDU Statechart Tools
& workspace setup

Setup

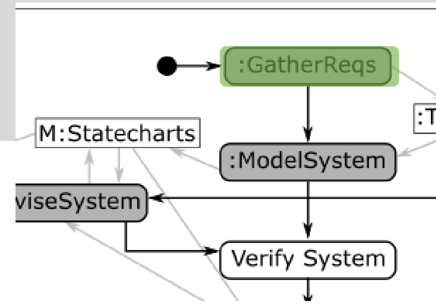
1. Prerequisite: you need an installed Java SDK
2. Install YAKINDU SCT
   1. Select the YAKINDU SCT archive for your OS on the stick and unzip it to any location (Windows Programs folder strongly not recommended)
   2. Alternatively download YAKINDU SCT from https://www.itemis.com/en/yakindu/state-machine/
3. Unzip the MODELS_SC_TUTORIAL.zip to any folder
4. Start YAKINDU SCT and select <tutorial folder>/workspace as workspace location when prompted.

-> YAKINDU SCT starts and you will see the welcome screen.

Exercise 1

Model a basic traffic light

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

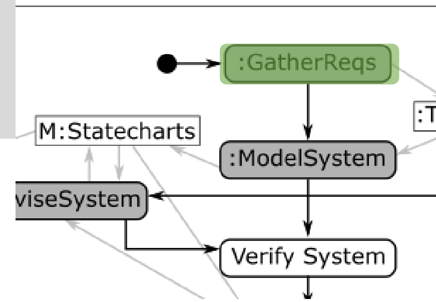
<<control>>

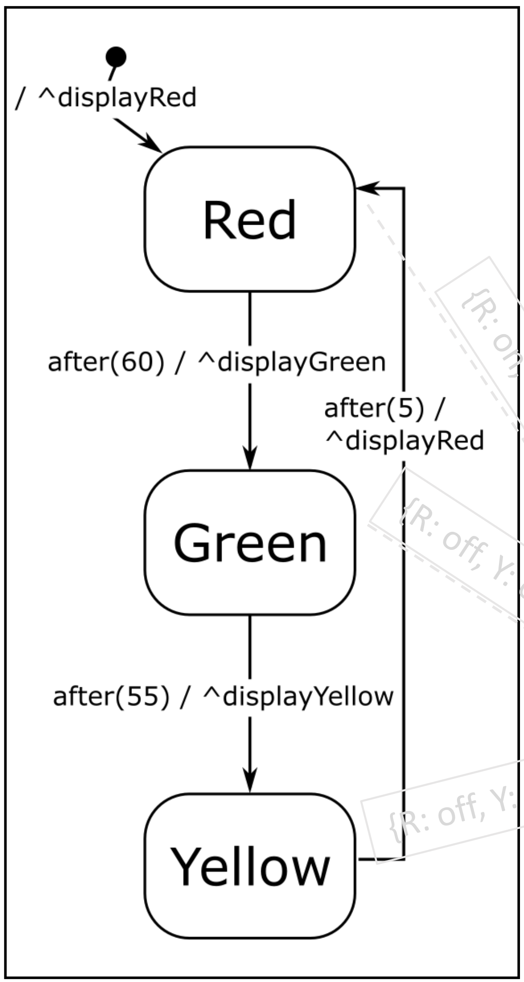
(Simulated) Plant

<<observe>>

Environment

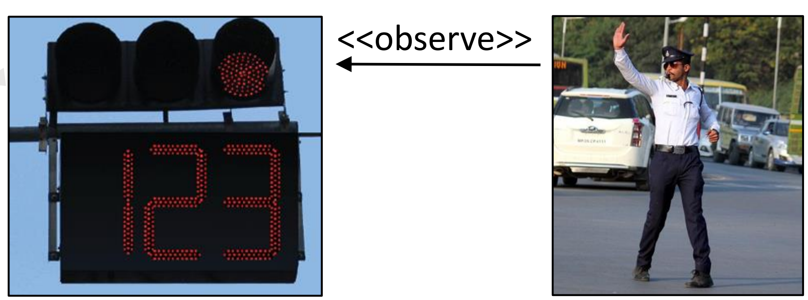
Exercise 1 - Solution



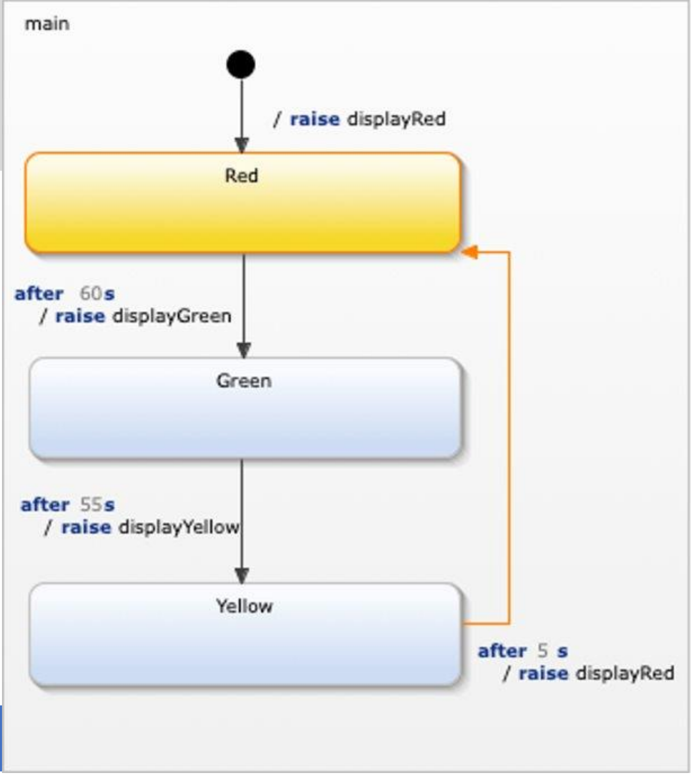


- R1: three differently coloured lights: red (R), green (G), yellow (Y)
- R2: at most one light is on at any point in time
- R3: at system start-up, the red light is on
- R4: cycles through red on, green on, and yellow on
- R5: red is on for 60s, green is on for 55s, yellow is on for 5s

(Simulated) Plant **Environment**



Exercise 1 - Solution

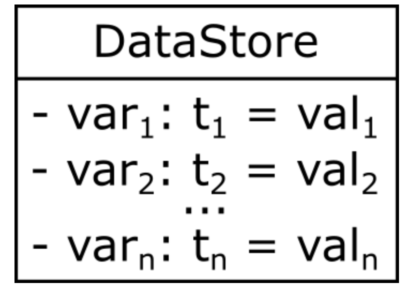


requirement	modelling approach
R1: three differently coloured lights: red (R), green (G), yellow (Y)	For each color a state is defined. Transitions that lead to a state raise the proper out event which interacts with the plant.
R2: at most one light is on at any point in time	The states are all contained in a single region and thus a exclusive to each other.
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

+



being **in** a state

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

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

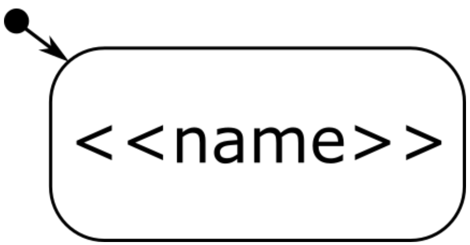
data store **snapshot**

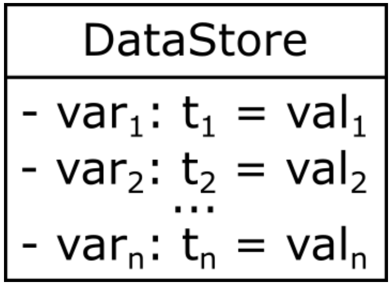
= variables and their value

=

full system state

Full System State: Initialization






```
int main() {

}
```

initial state

exactly one per model

“entry point”

provide **default value** for each variable

“initial snapshot”

Compare:

C++ initialization

implicit state

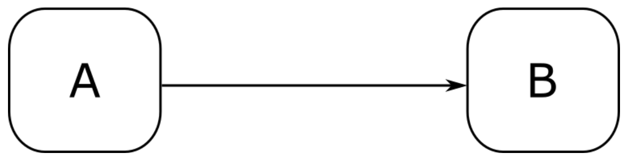
(program counter)

+ **data store**

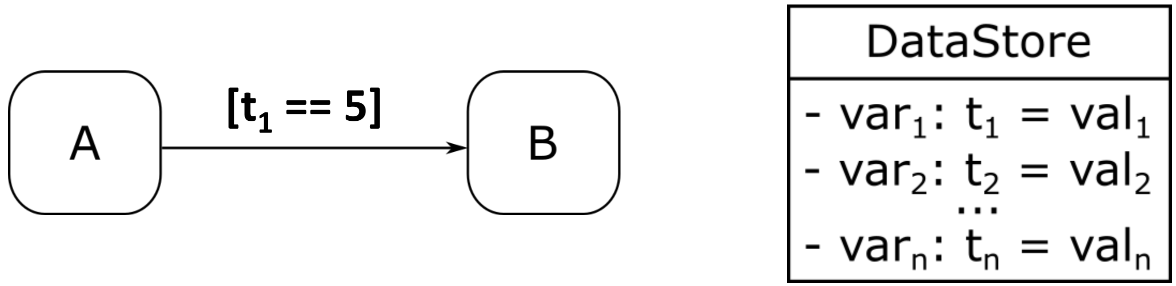
Transitions: Guards

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

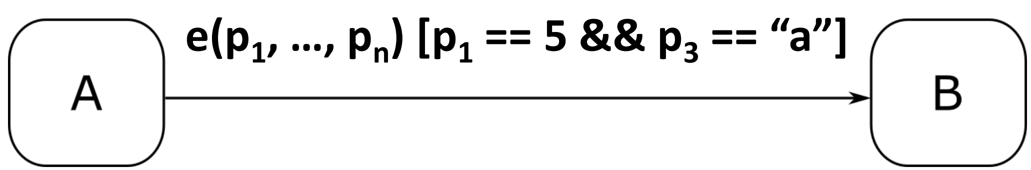
- Modelled by action code in some appropriate language
- Spontaneous



- Data Store Variable



- Parameter Variable

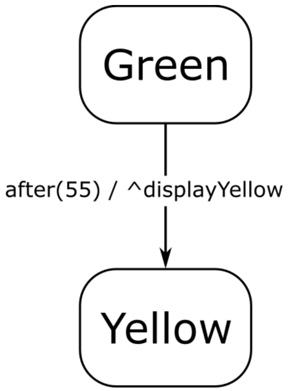


Transitions: Output Actions

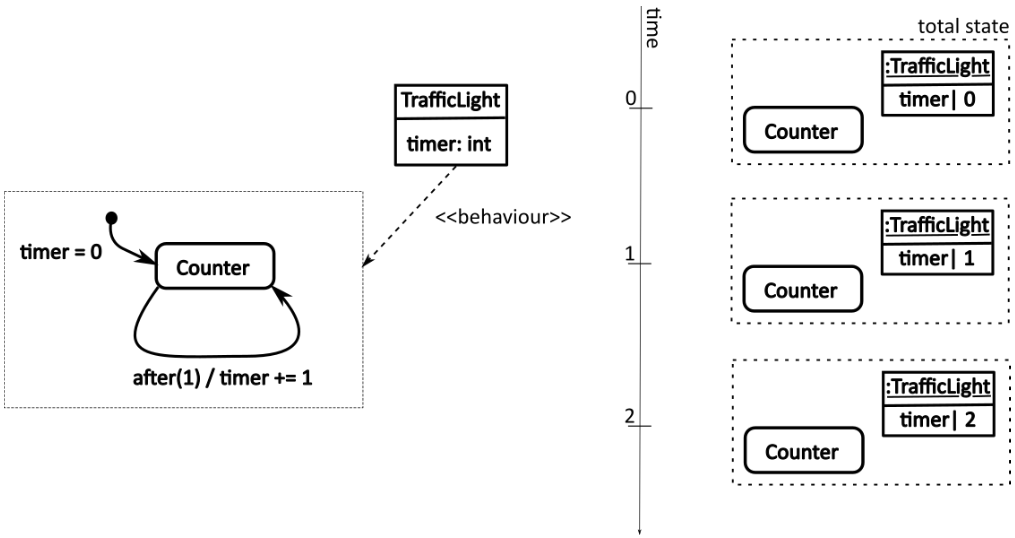
event(params) [guard] / output_action(params)

- Output Event

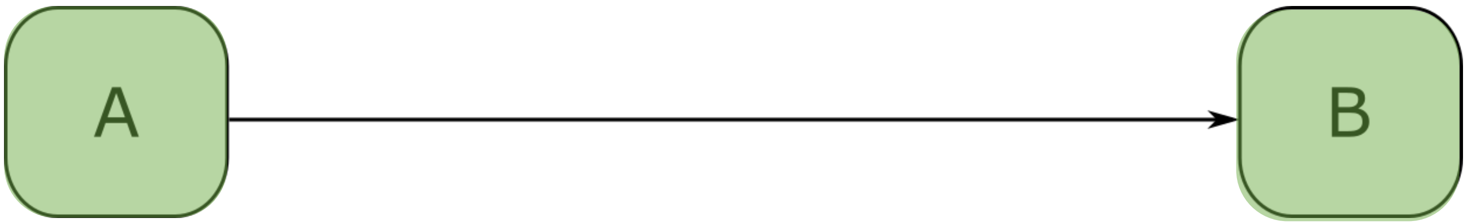
  ^*output_event*(p_1, p_2, …, p_n)



- **Assignment** (to the non-modal part of the state)
  - by action code in some appropriate language

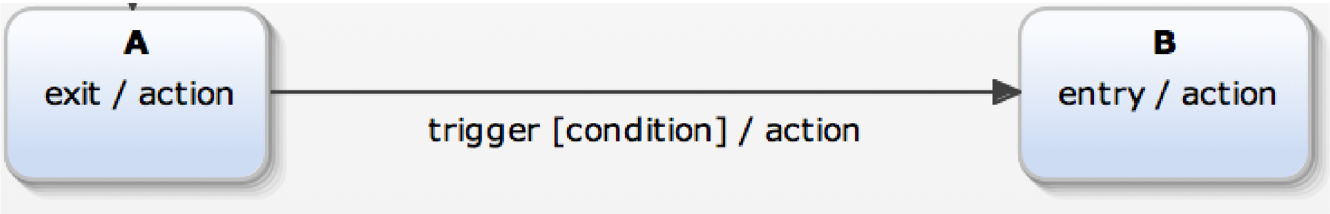


Transitions



- Model the **dynamics** of the system:
  - *if*
    - the system is **in state A**
    - and **event is processed**
    - and **guard** evaluates to **true**
  - *then*
    1. **output_action** is evaluated
    2. and the new **active state** is B

Transition Execution




```
if  A is active
{
   if (
         ((trigger specified AND occurred) OR (no trigger specified))
          AND
          ((condition specified AND is true) OR (no condition specified))
      )
   {
      exit A
      execute exit action
      execute transition action
      execute entry action
      enter B
      }
}
```

YAKINDU Variables

Variables & Constants

- In statecharts variables hold quantitative values.
- Variables may be accessible from ‚outside‘ the statechart
- Variables behave like you would expect

```
// a simple variable
var x : integer

// ... with initialization
var x : real = 4.2
var z : boolean = true
var my_var : integer = 0xff
```

- You can define constants
- Constants must have an initial value

```
const PI : real = 3.14
```

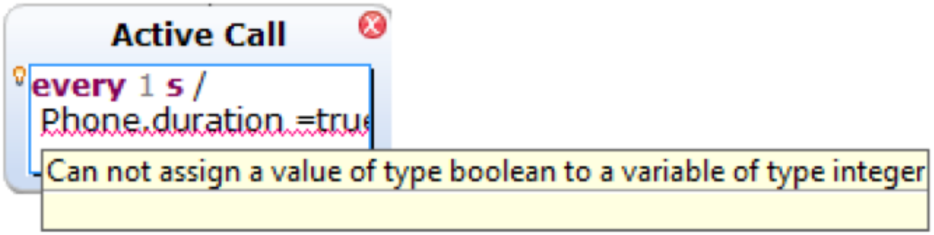
State and Statechart Trigger

The following trigger types can be used within state and statechart specifications:

```
buttonPressed      // named event triggers

after 2s           // one shot time trigger
every x ms         // periodic time trigger

always             // pseudo trigger that is always active
oncycle            // same as ‚always‘

entry              // pseudo trigger for entry actions
exit               // pseudo trigger for exit actions
```

Type System

- YAKINDU provides a 'Pluggable Type System'
- Type contributions can be provided by plugins
- Type checking is performed on expressions.
- Type inferrercalculates (infers) types of expressions and checks type constraints.
- Type inferreris extendable in order to implement specific checks.
- Immediate user feedback while editing



Type Systems

Simple type system provides:

- boolean
- integer
- real
- string
- void

Complex type system support "deep integrations" with modeling and programming languages (DSLs, Franca IDL, C, C++, Java, TypeScript):

- Custom types
- Structured types
- Enumerations
- Generics
- etc..

Variable Usage Scenarios

Internal state

- hold internal values not visible outside

Parametration

- makes properties publicly accessible

Data Flow

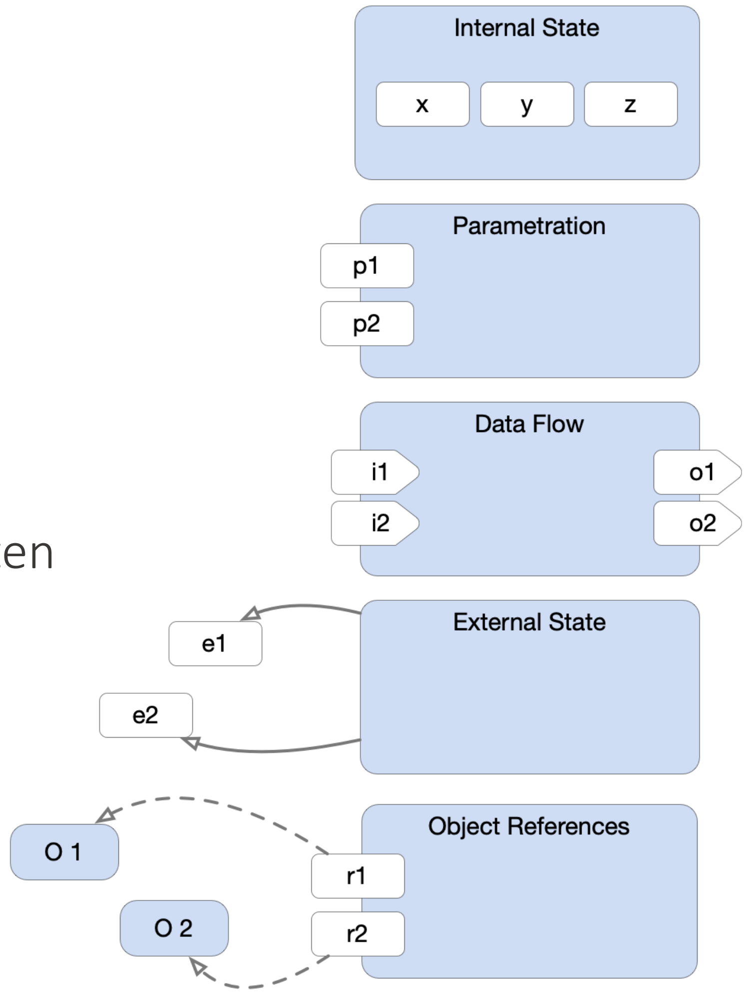
- 'in' and 'out' variables - in only read, out only written

Access external state

- externally declared variables can be in scope

Access objects

- variables can hold references to data & objects



C-like Expressions

operators

- assignment:
  `=, +=, -=, *=, /=, %=, <<=, >>=, &=, ^=, |=`
- boolean:
  `&&, ||, !`
- compare:
  `==, !=`
- arithmetic:
  `+, -, *, /, %`
- bit:
  `&, |, ^, >>, <<`
- ternary: `()?:`

literals

- decimal, hex & binary integers
- floating point
- boolean (true, false)
- string

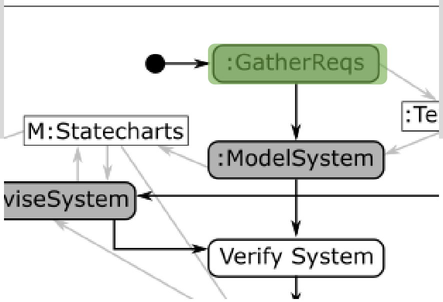
other

- `active(`*`statename`*`)`
  checks if the specified state is active
- `valueof(`*`event`*`)`
  gets the payload of the event
- `as`
  type cast

Exercise 2

Add data stores

Exercise 2 - Requirements



- R6: In the last 6 seconds of red being on, the light prepares to go to green by blinking its yellow light (1s on, 1s off) in addition to its red light being on.
- R7: The time period of the different phases should be configurable.

Your model here.

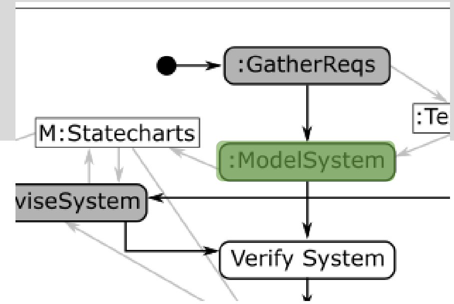
TrafficLight
- counter: Integer = 0 - green: Boolean = false - red: Boolean = false - yellow: Boolean = false

<<behavior>>

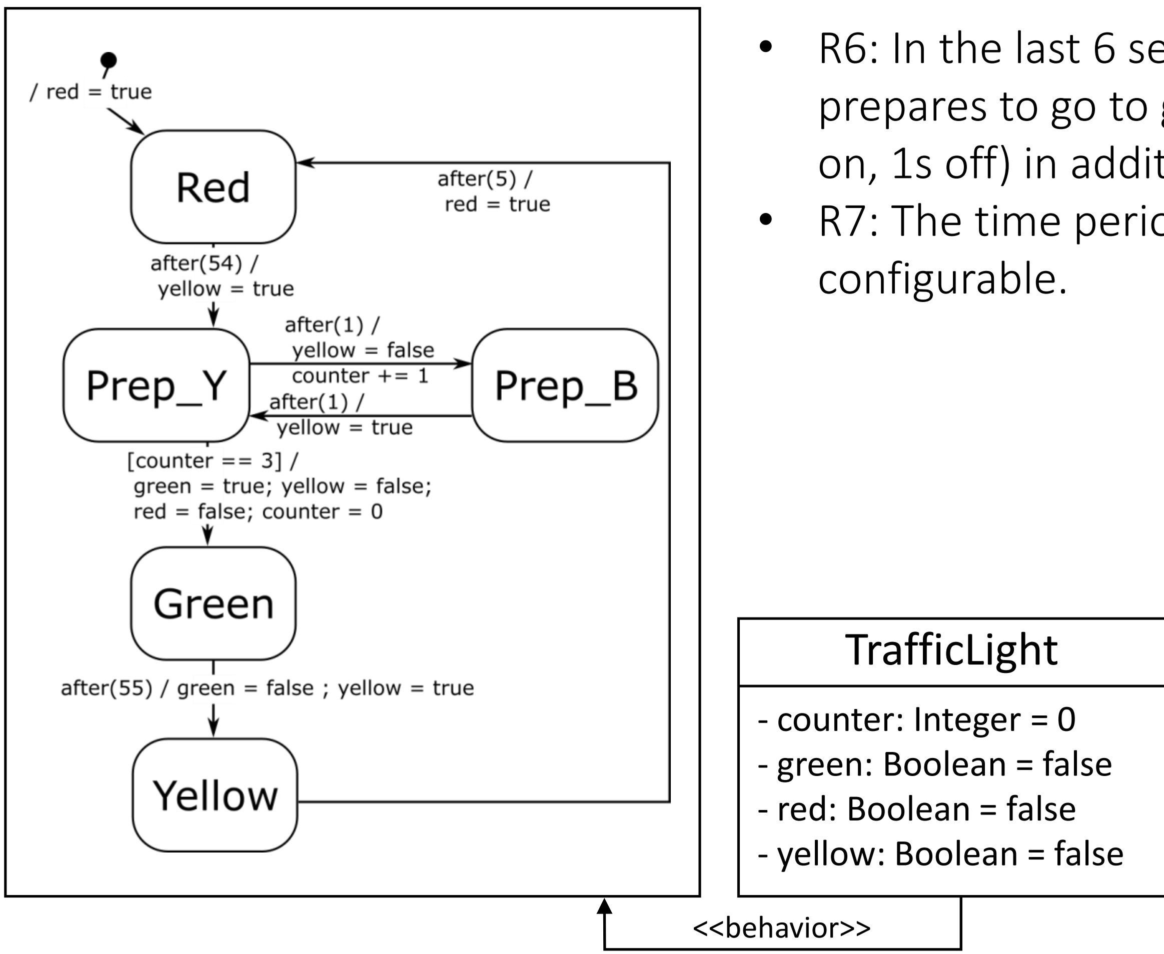
Make sure that:

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

Exercise 2: Solution



- R6: In the last 6 seconds of red being on, the light prepares to go to green by blinking its yellow light (1s on, 1s off) in addition to its red light being on.
- R7: 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 transforms the state machine from a valid state configuration into the next valid state configuration.
- RTC steps are executed one after each 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) {

}
```

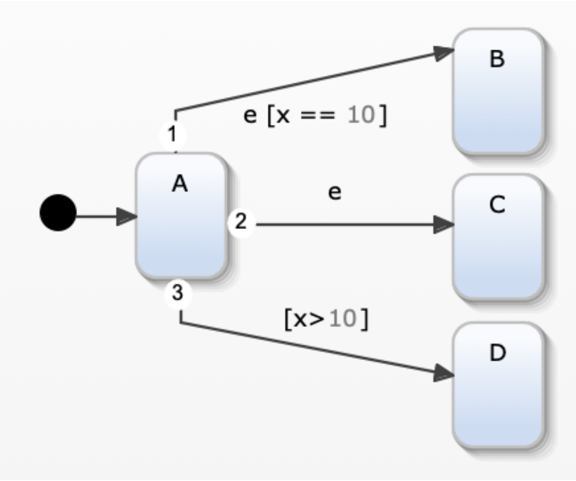
YAKINDU strategy: first enabled transition is selected. If found no further transitions are tested.

Enabled:

```
if (
    ((trigger specified AND occurred) OR (no trigger specified))
     AND
     ((condition specified AND is true) OR (no condition specified))
  )
```

First:

transitions are ordered – first according to this order



Flat Statecharts: Simulation Algorithm (2)

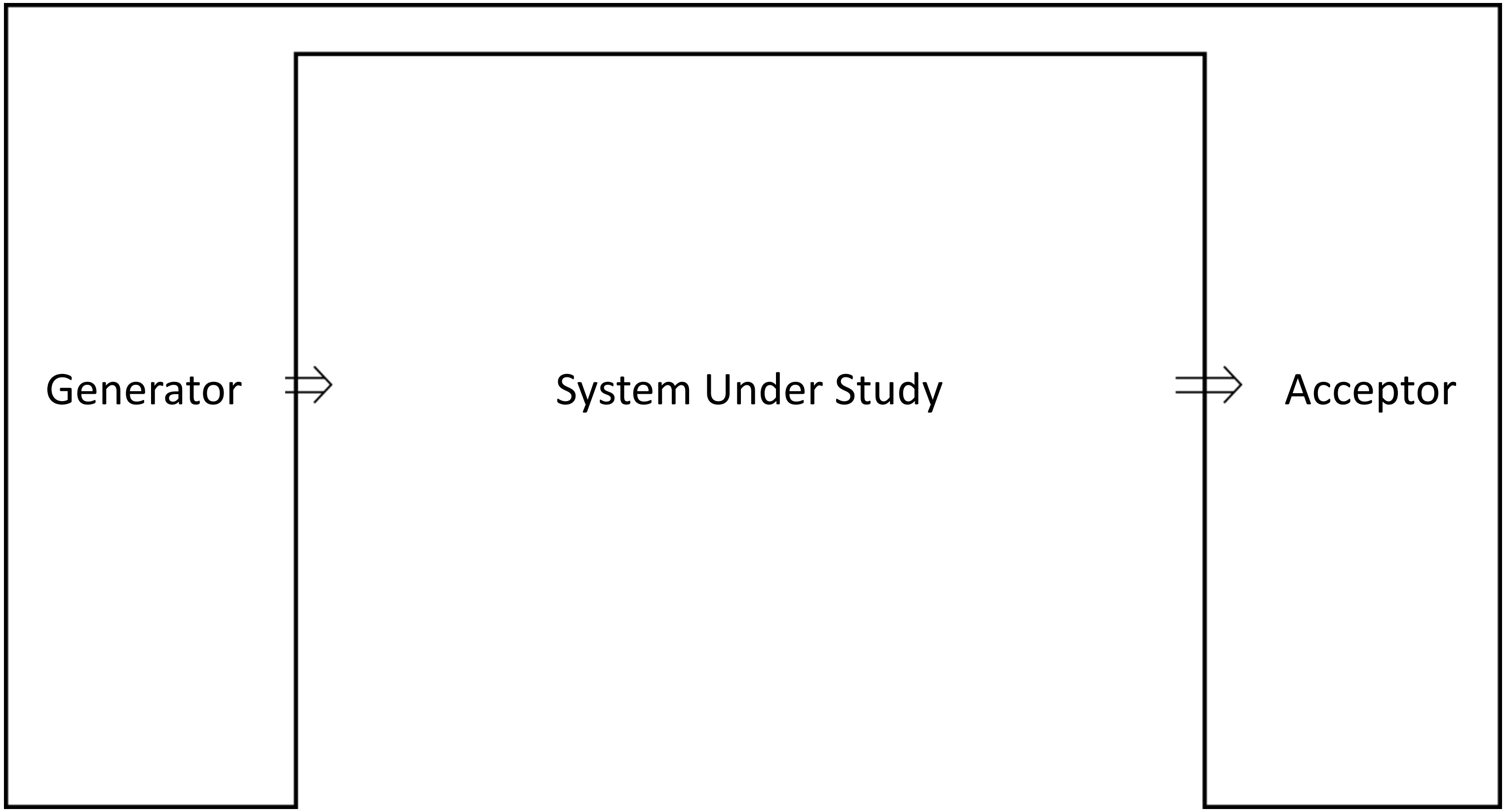
```
simulate(sc: Statechart) {
    input_events  = initialize_queue()
    output_events = initialize_queue()
    timers        = initialize_set()
    curr_state    = sc.initial_state
    for (var in sc.variables) {
        var.value = var.initial_value
    }
    while (not finished()) {
        curr_event = 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

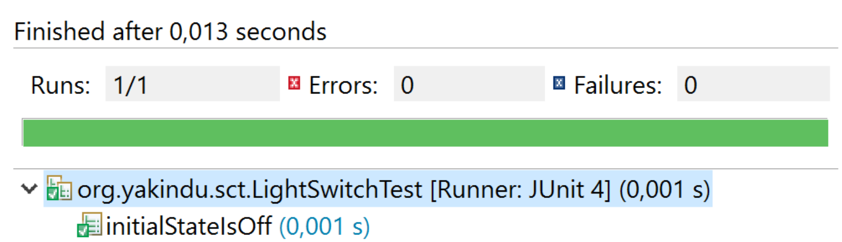


Zeigler BP. Theory of modelling and simulation. New York: Wiley-Interscience, 1976.

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.

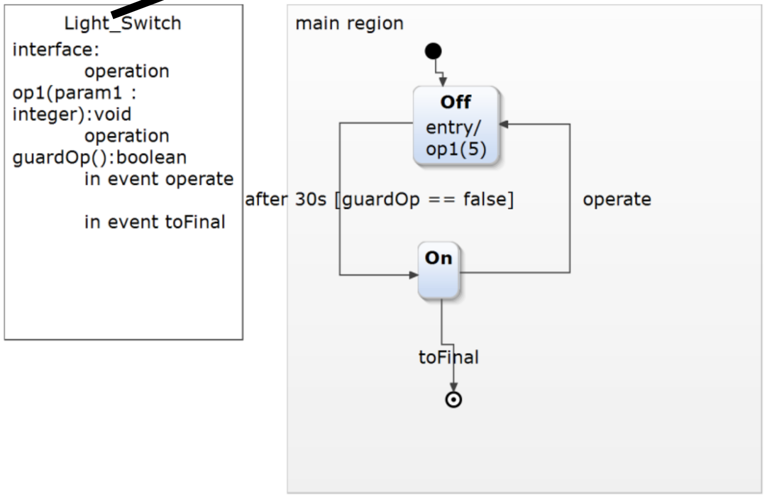
SCTUnit (beta)

- 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




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

```
testclass someTestclass for statechart Light_Switch {

}
```



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

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

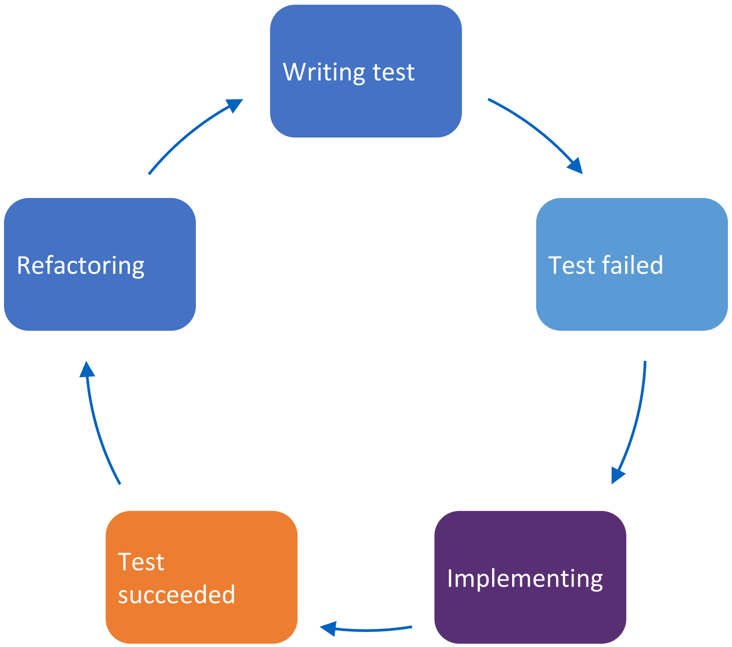
```
// mocking the return value of an operation
mock mockOperation returns (20)
mock mockOperation(5) returns (30)

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

```
// if expression

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

```
// while expression

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

Test-Driven Development

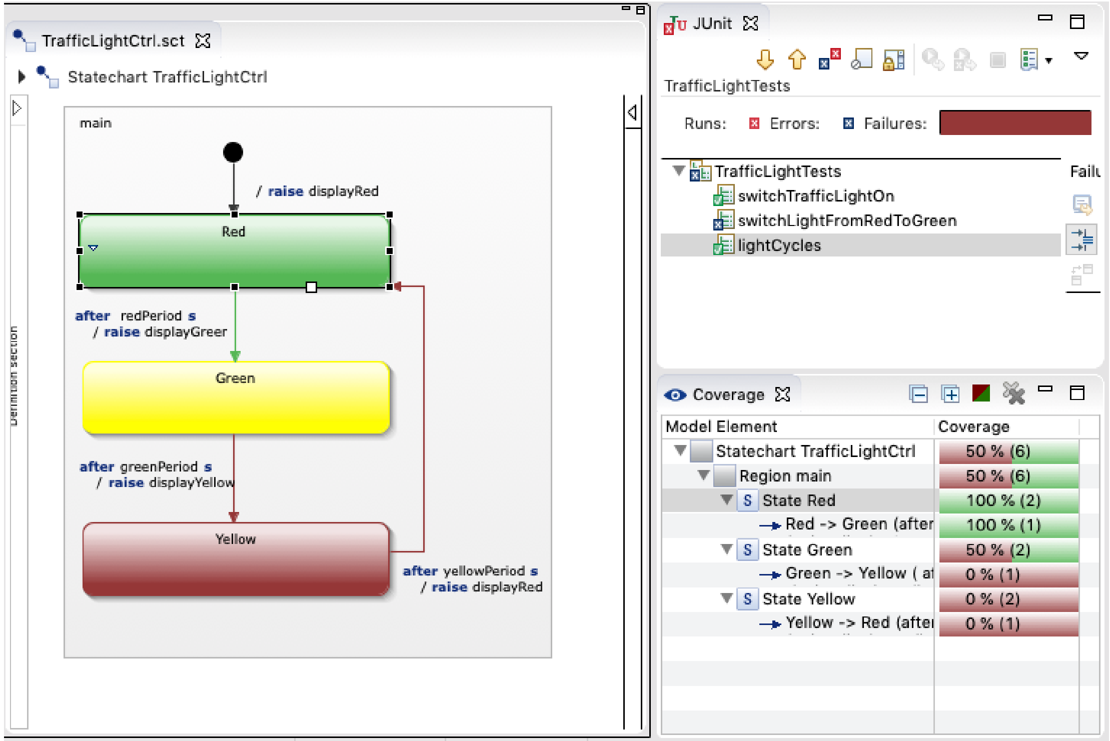
- Software development process, where software is developed driven by tests
- Test-first-approach
- 3 steps you do repeatedly:
    - writing a test
    - implementing the logic
    - refactoring



Exercise 3

Testing Models

Exercise 3 – Unit testing statecharts



- Run and inspect prepared tests
- Fix and complete tests
  - make them green
  - 100% coverage

Generating Code (1)

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 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

Exercise 4

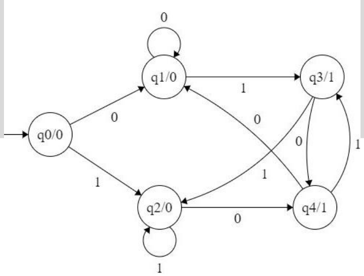
Generating Code

Exercise 4 – Integrate generated code with UI

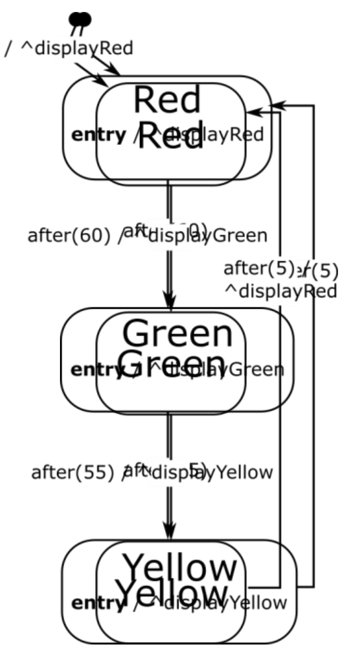
- Inspect the code generator model
- Inspect the generated code
- Integrate the state machine
- Run the UI

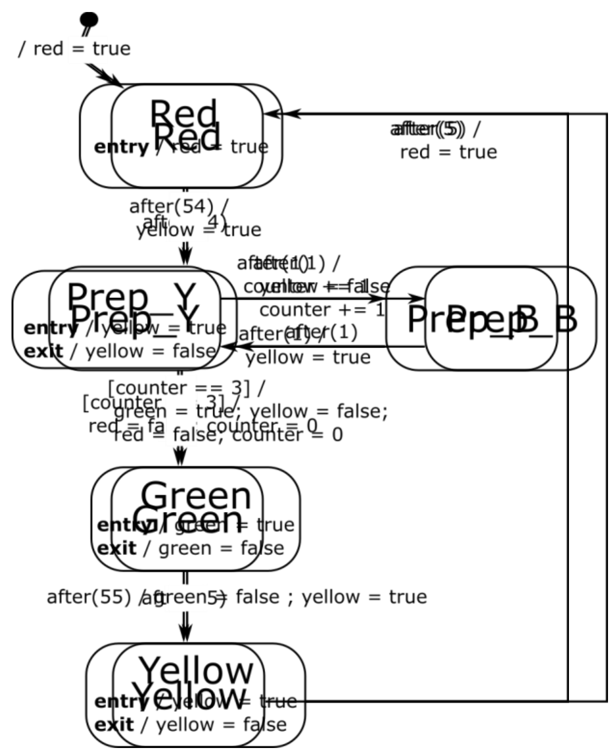
Hierarchy

Entry/Exit Actions

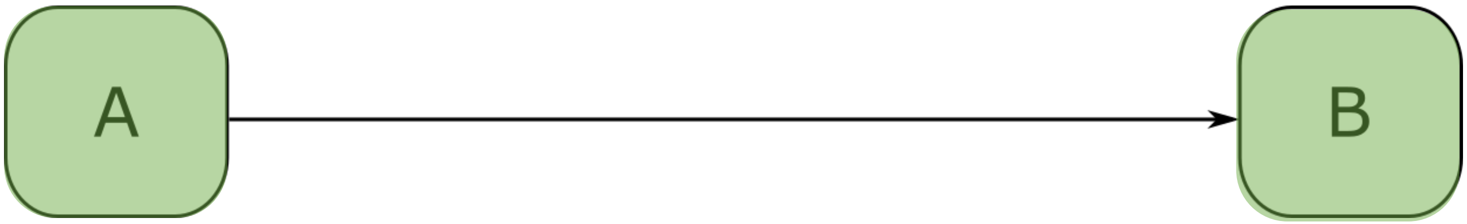


- A state can declare entry and exit actions.
- An *entry action* is executed whenever a state is entered (made active).
- An *exit action* is executed whenever a state is exited (made *inactive*)
- Same expressiveness as *transition actions:*





Transitions



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

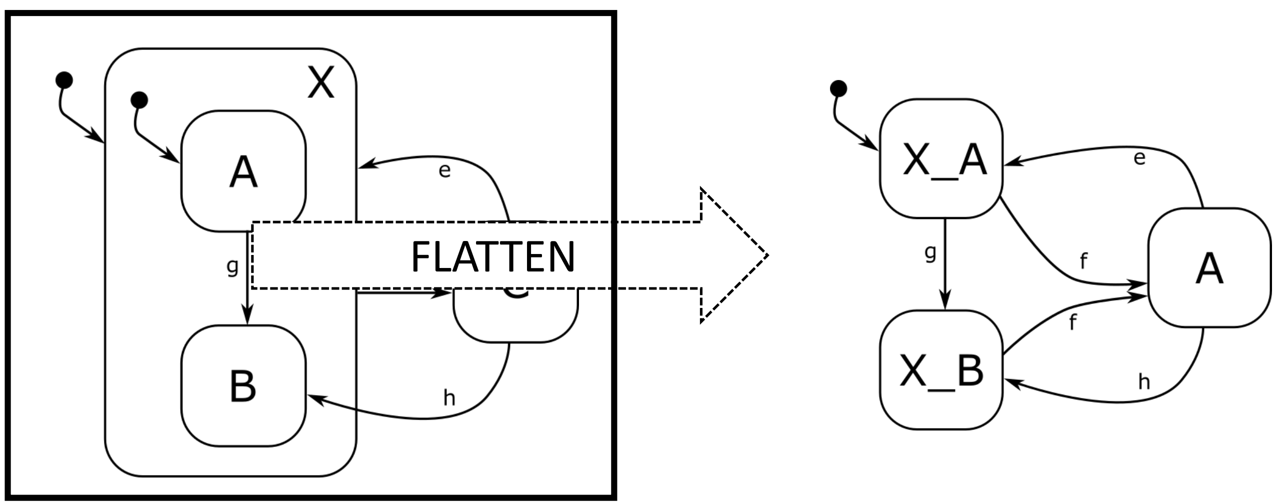
Entry/Exit Actions: Simulation Algorithm

```
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)
        }
    }
}
```

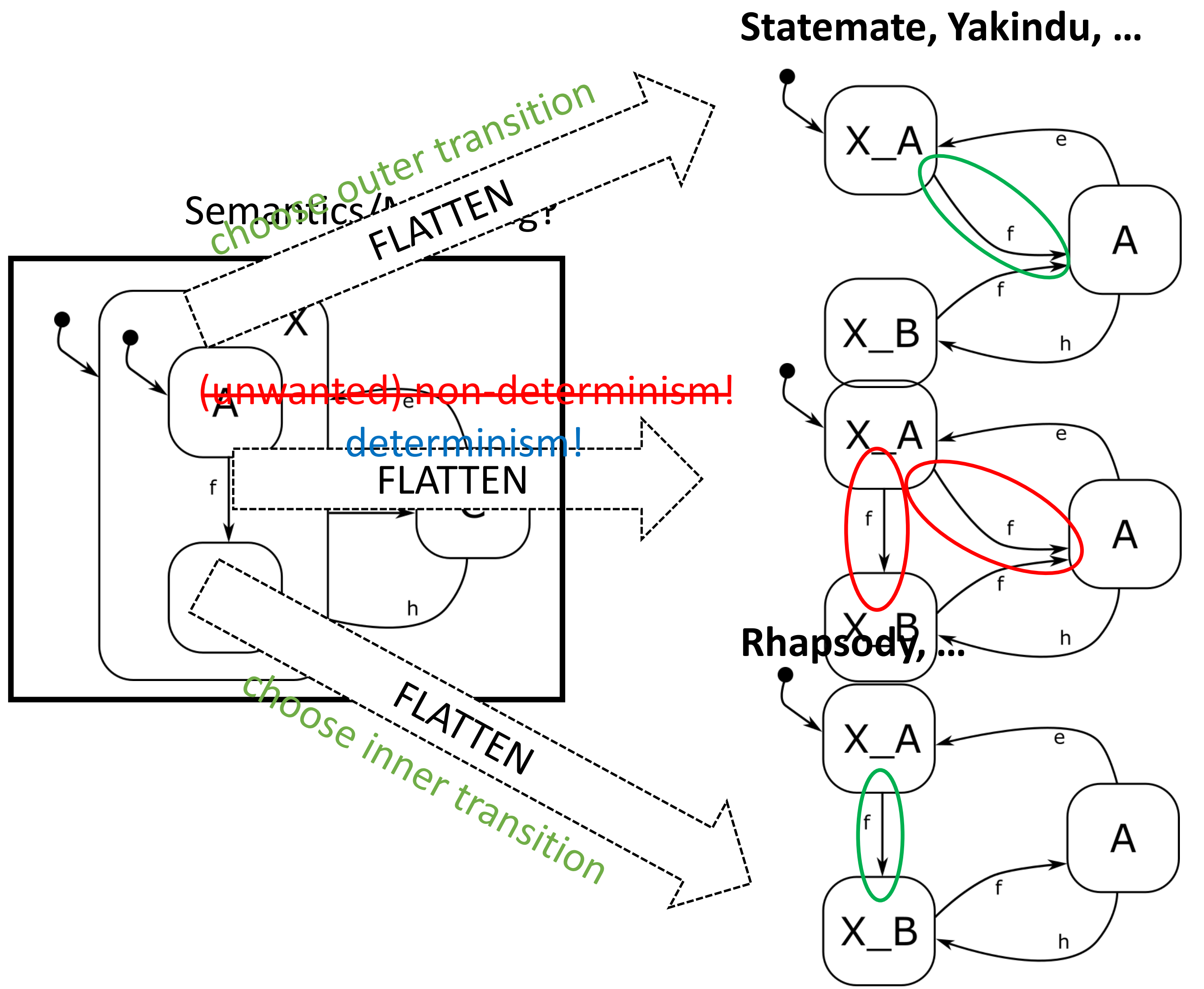
Hierarchy

- Statechart states can be hierarchically composed
- Each hierarchical state has exactly one initial state
- An active hierarchical state has exactly one active child (until leaf)

Semantics/Meaning?

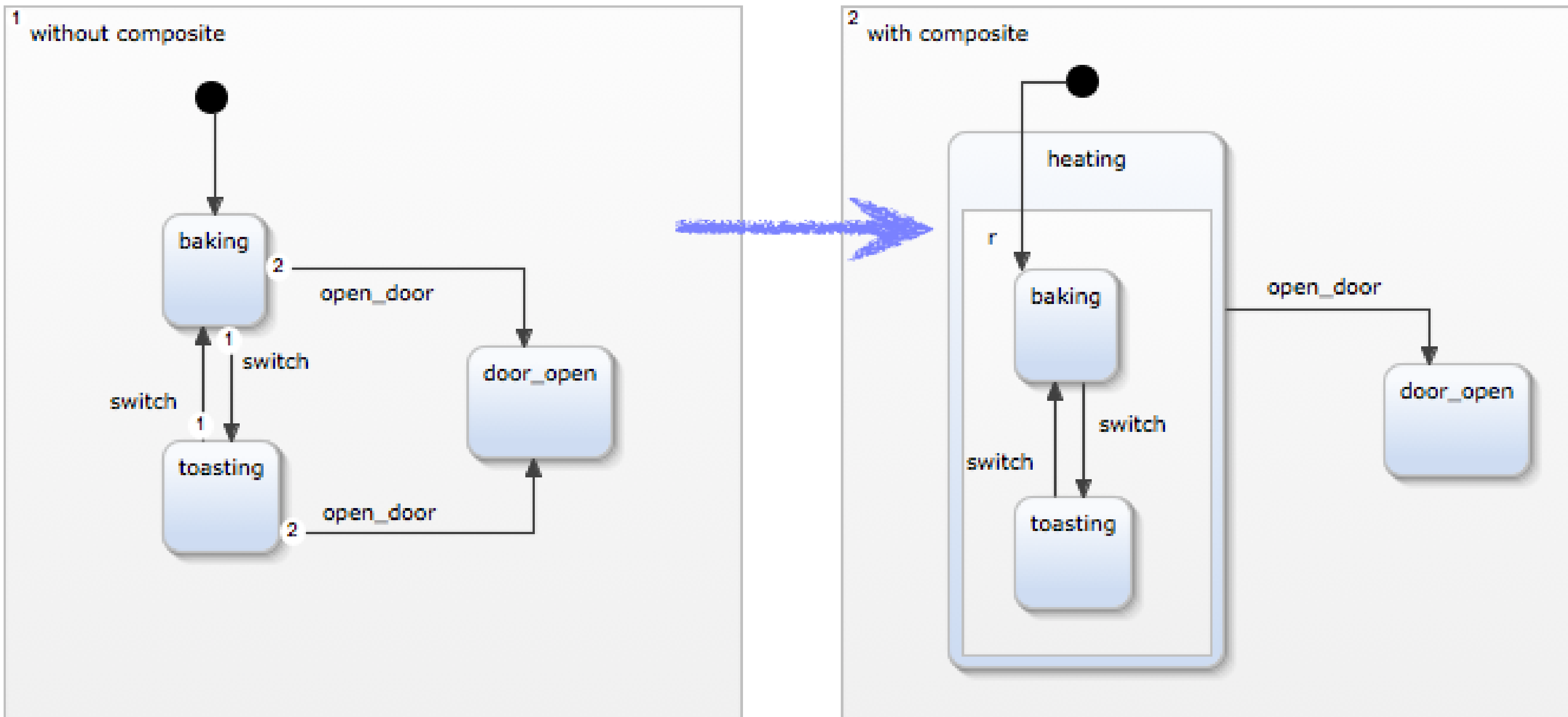


Hiearchy: Modified Example

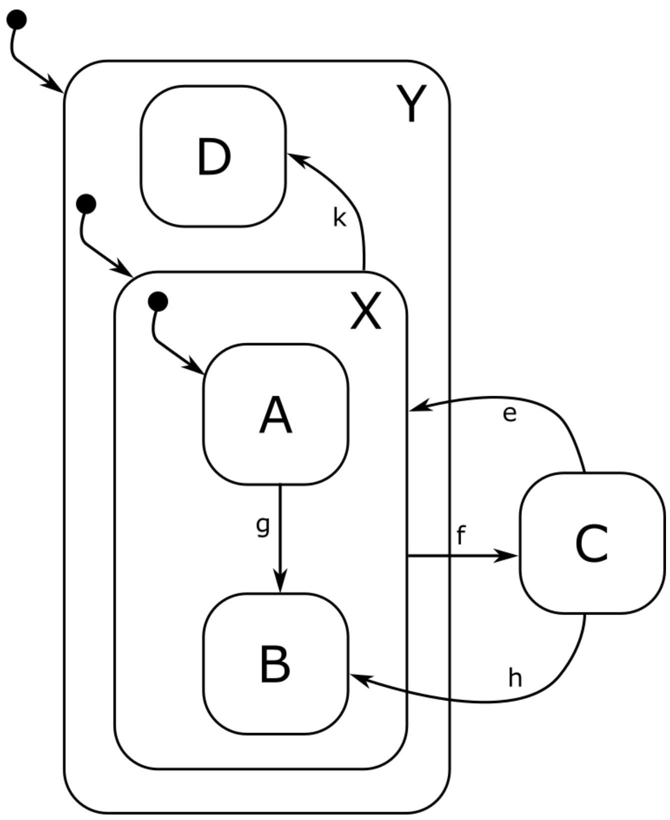


Composite States

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

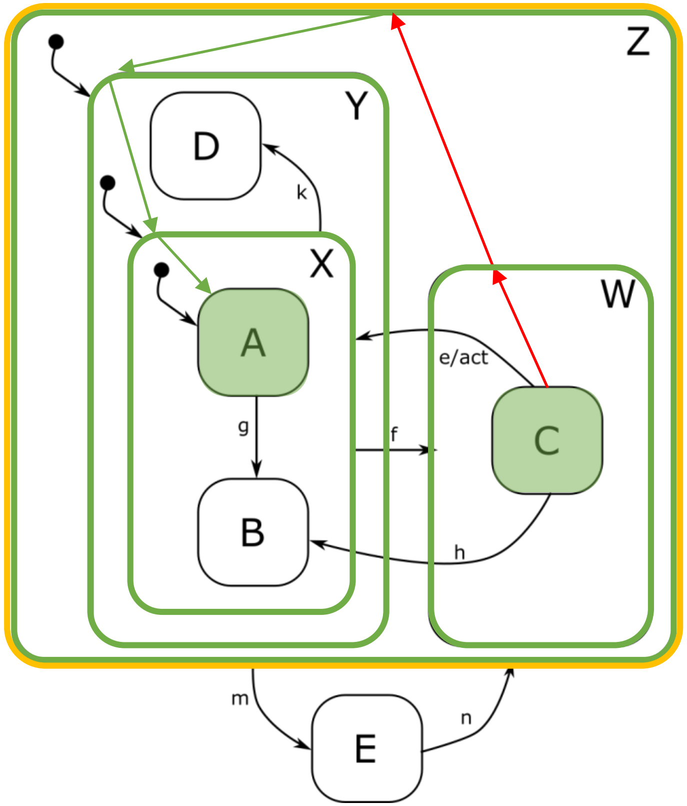


Hierarchy: Initialization



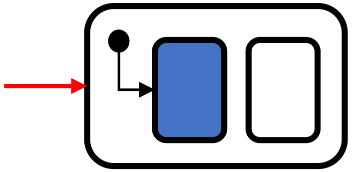
- Concept of *effective target state*
  - Recursive: the effective target state of a composite state is its initial state
- Effective target state of initial transition is *Y/X/A*
- Initialization:
  1. Enter Y, execute enter action
  2. Enter X, execute enter action
  3. Enter A, execute enter action

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:

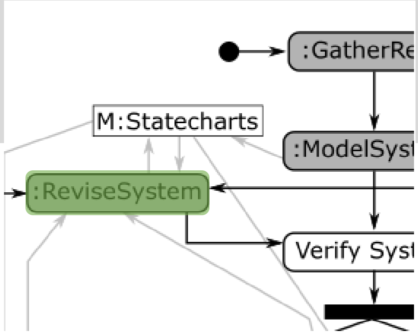


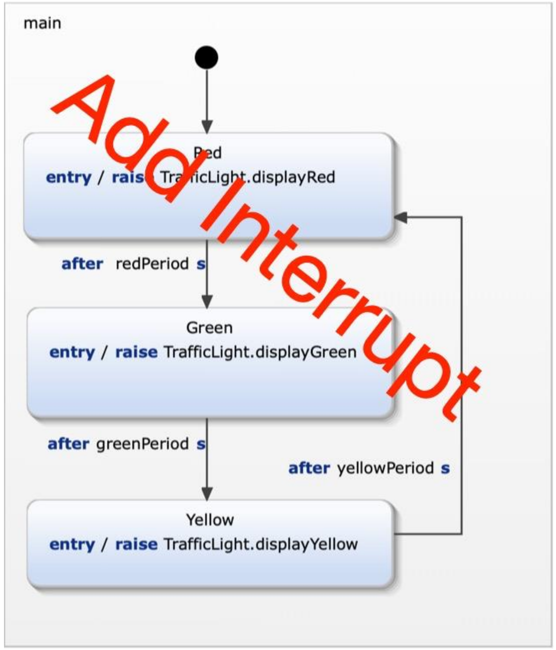
RECURSIVE!

Exercise 5

Model an interruptible traffic light

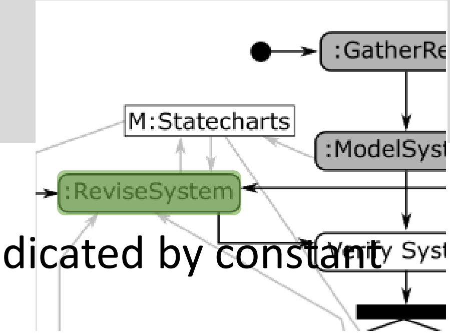
Exercise 5 - Requirements



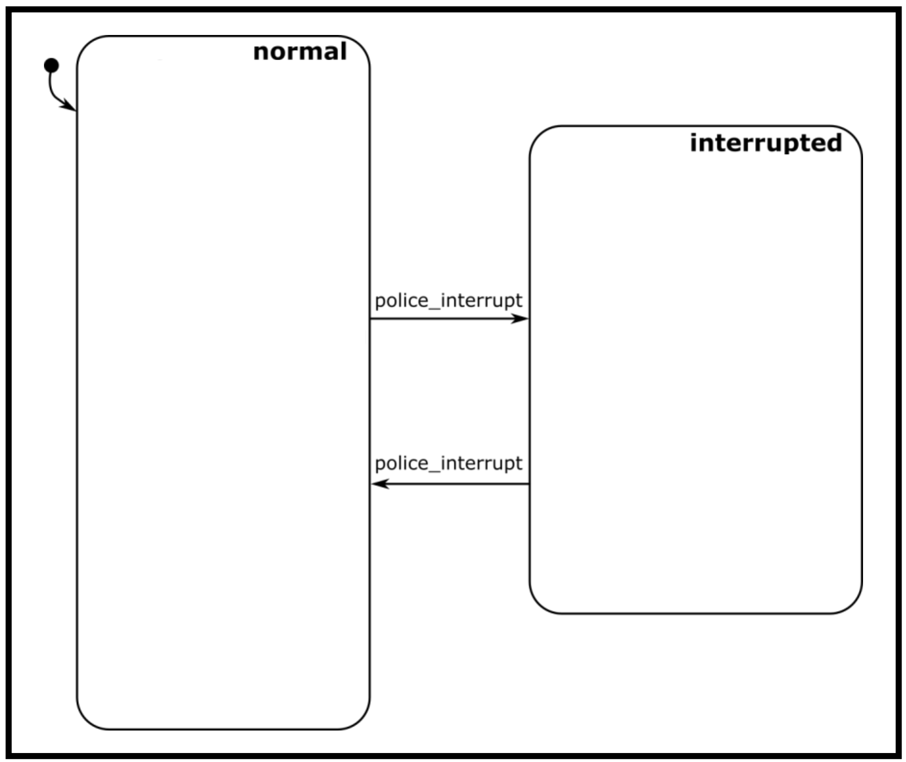


- R7a: police can interrupt autonomous operation .
- R7b: Autonomous opreration can be interrupted during any pahse indicated by constant red, yellow and green lights.
- R7c: In interruptetd mode the yellow light blinks with a constant frequency of 1 Hz. (on -> 0.5s, off 0.5s).
- R8a: Police can resume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

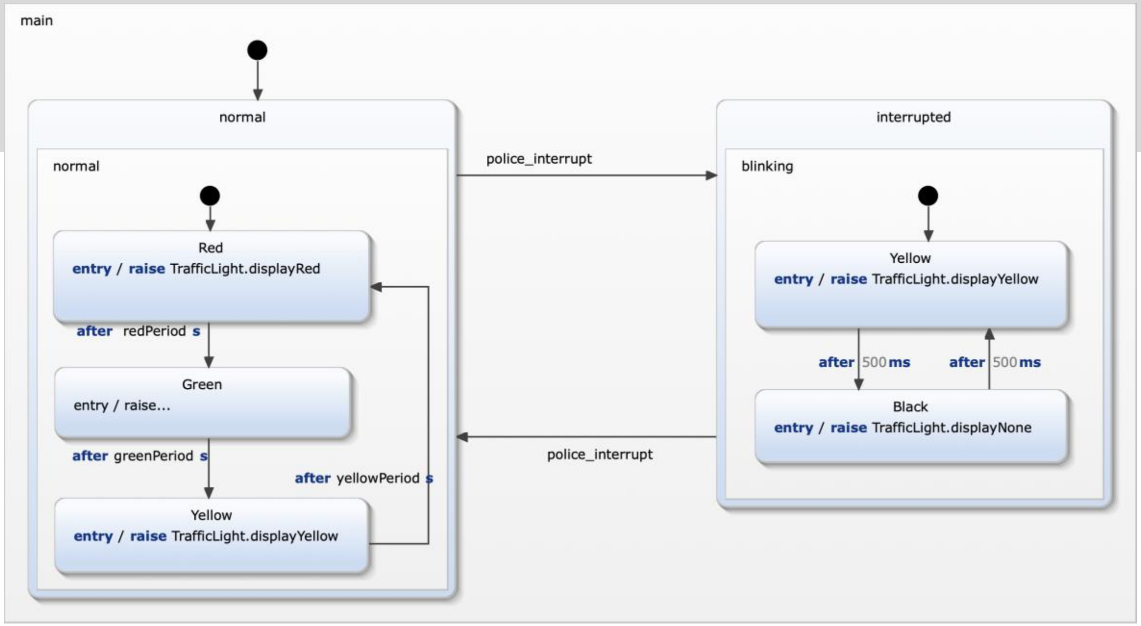
Exercise 5: Solution



- R7a: police can interrupt autonomous operation .
- R7b: Autonomous opreration can be interrupted during any pahse indicated by constant red, yellow and green lights.
- R7c: In interruptetd 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

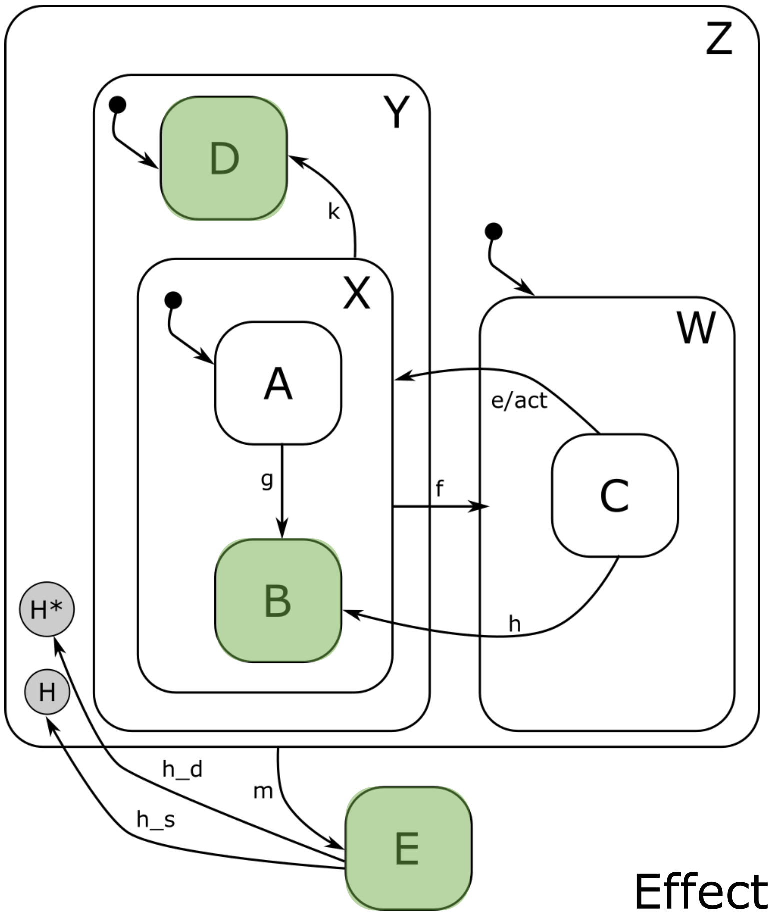


requirement	modelling approach
R6: police can interrupt autonomous operation.	An new incoming event police_interrupt triggers a transition to a new state interrupted.
R6a: Autonomous opreration can be interrupted during any pahse indicated by constant red, yellow and green lights.	The states Red, Green, 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

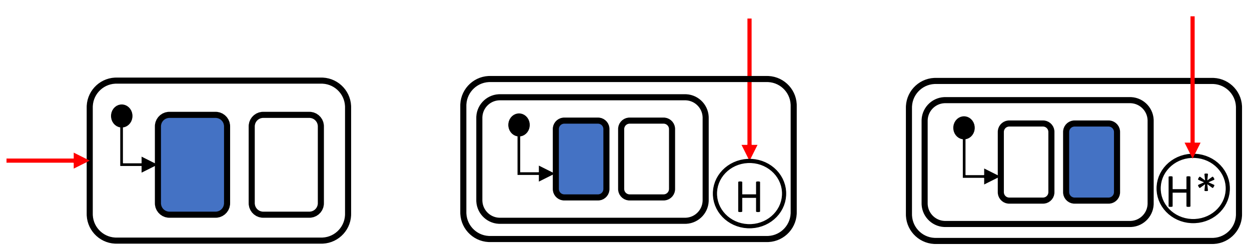
History

(H) *shallow* history (H*) *deep* history



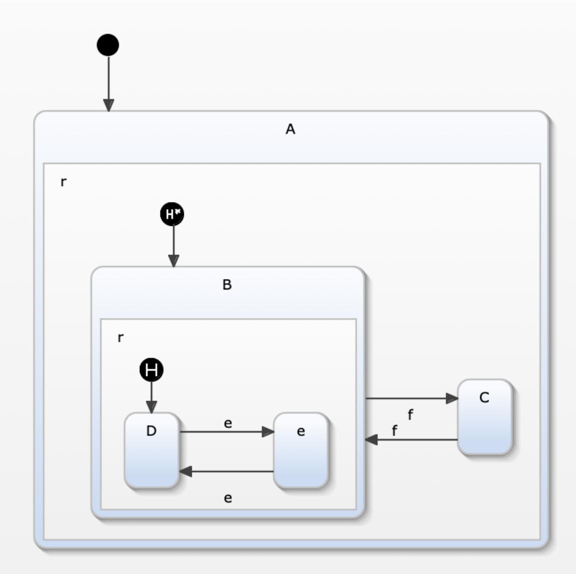
- Assume *Z/Y/X/B* is active, and *m* is processed
  - Effective target state: *E*
- If *h_s* is processed
  - Effective target state: *Z/Y/D*
- If *h_d* is processed
  - Effective target state: *Z/Y/X/B*

Effective target states:

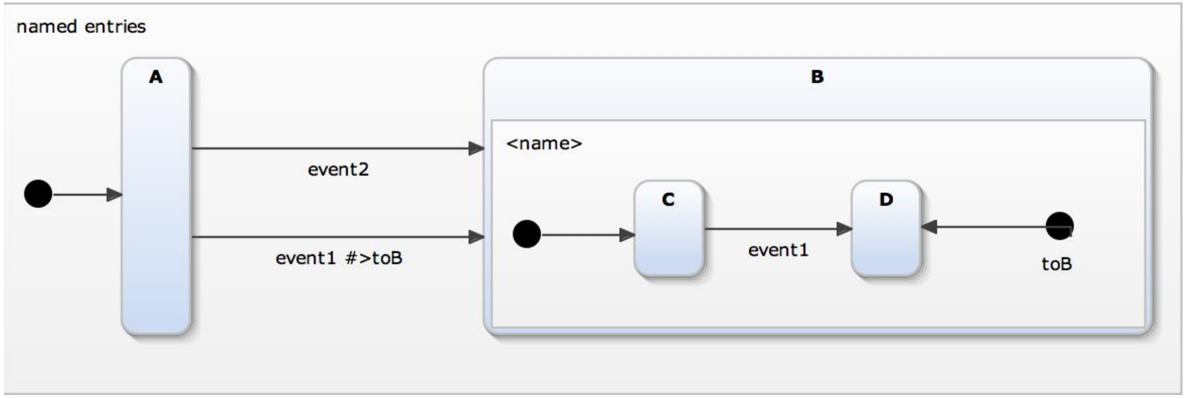


RECURSIVE!

- Entry- and exit-nodes define, how regions are entered or exited.
- There are three kind of entry nodes (initial, shallow history, deep history), but just one exit node.

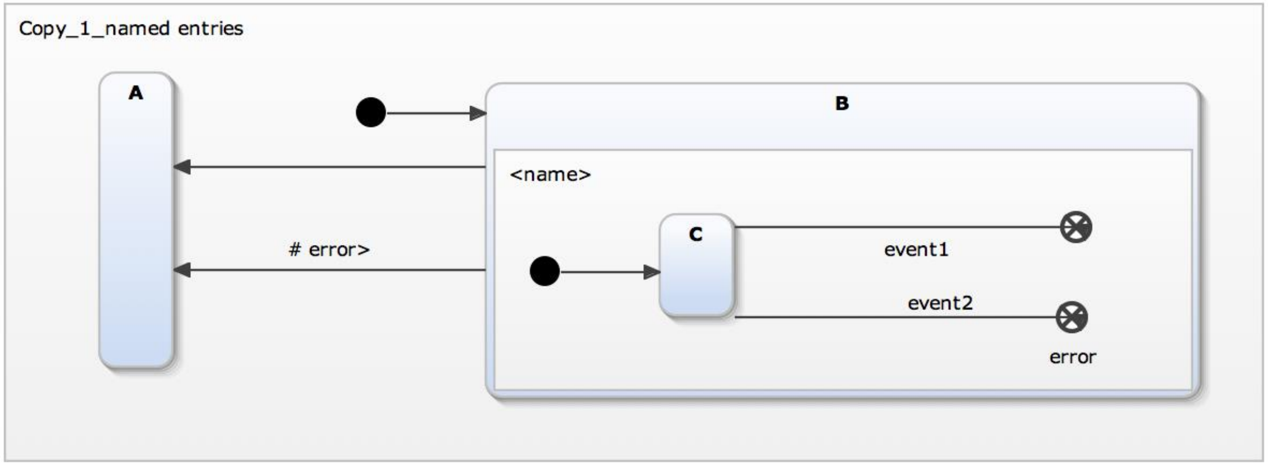


Named Entries



- Named entry nodes work like „go to“.
- The transition A>B trigger by event1 will enter C.
- The transition A>B trigger by event2 will enter D through the named entry ‚toB‘.

Named Exits

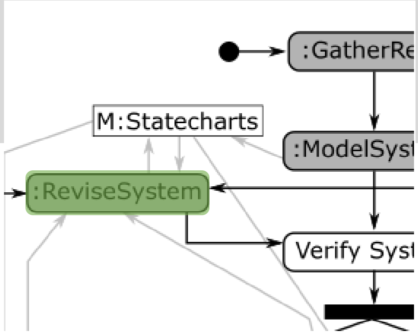


- Named exit nodes work like „come from".
- The upper transition B>A will be taken on event1.
- The lower transition B>A will be taken on event2 through named entry ‚error'

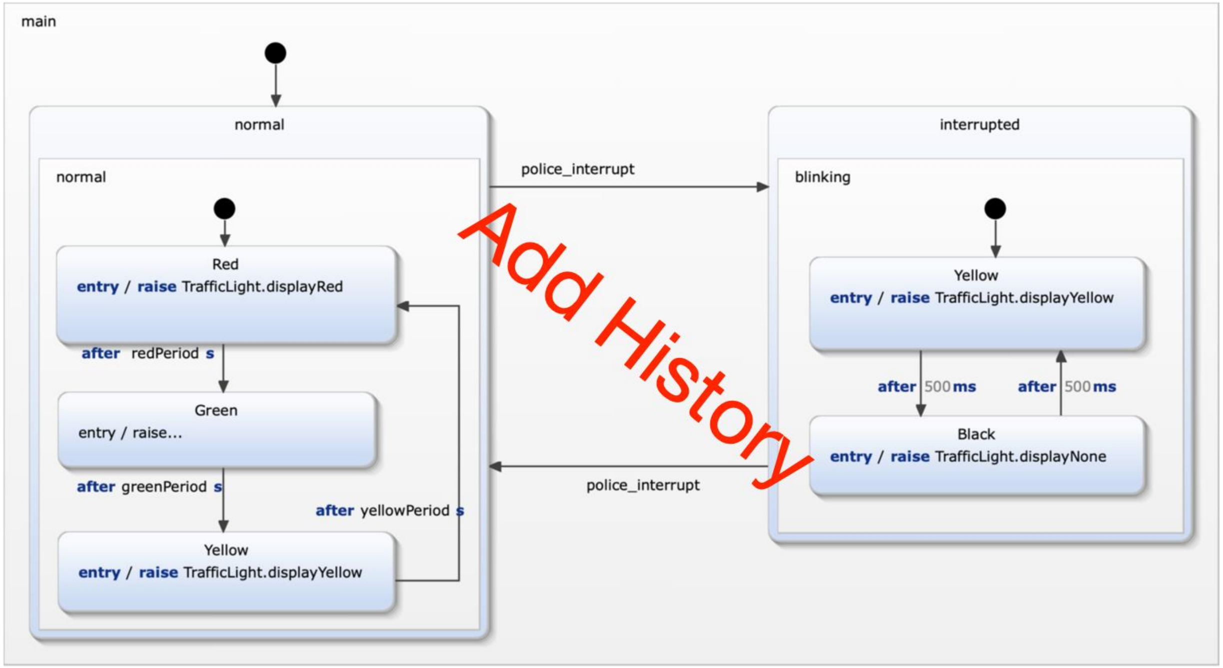
Exercise 6

Model an interruptible traffic light that restores its state

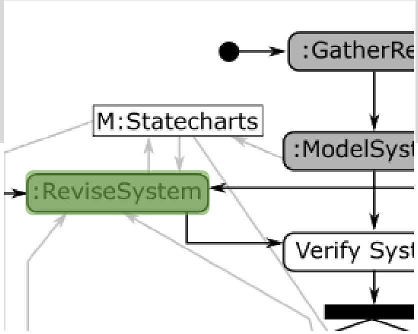
Exercise 6: Requirements



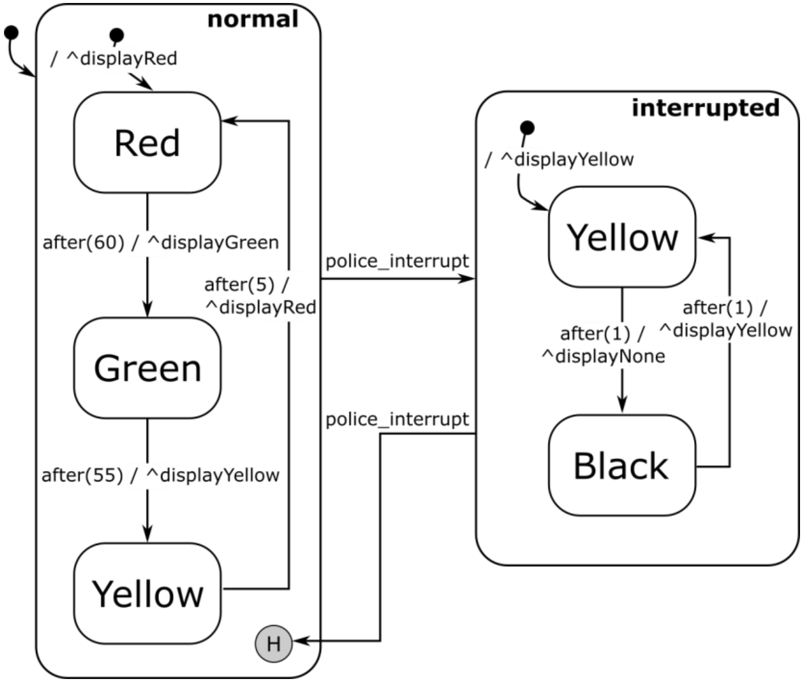
- R8b: when regular operation is resumed the traffic light restarts with the last active light color red (R), green (G), or yellow (Y) on.



Exercise 6: Solution



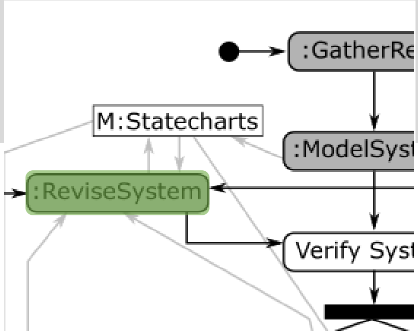
- R8b: when regular operation is resumed the traffic light restarts with the last active light color red (R), green (G), or yellow (Y) on.



Exercise 7

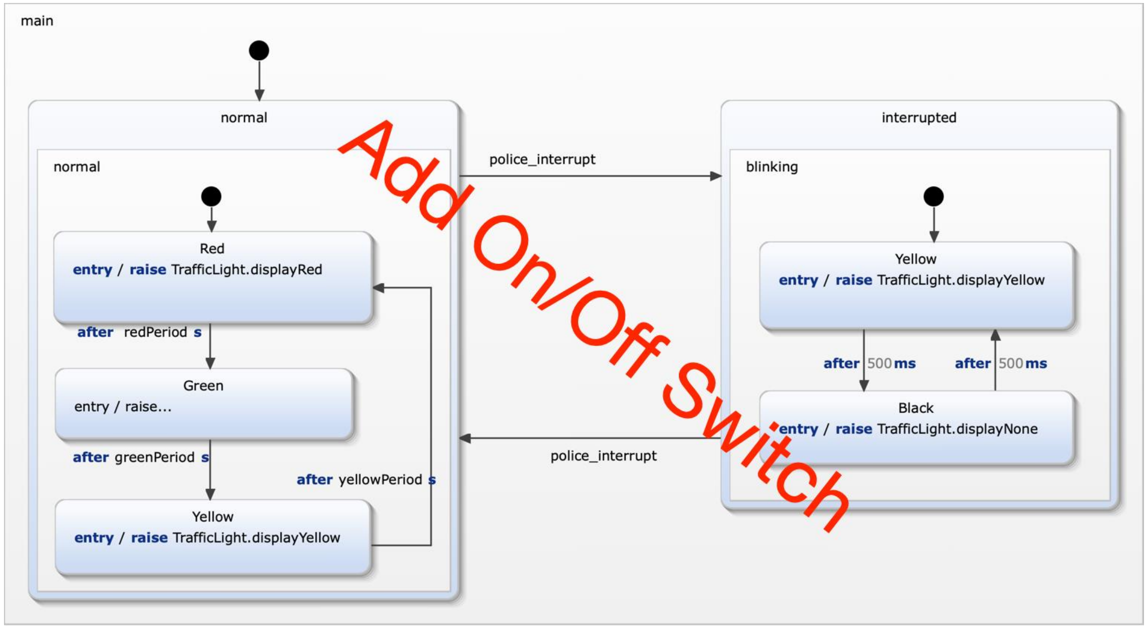
Model an interruptible traffic light that restores its state and can be switched on/off

Exercise 7: Requirements

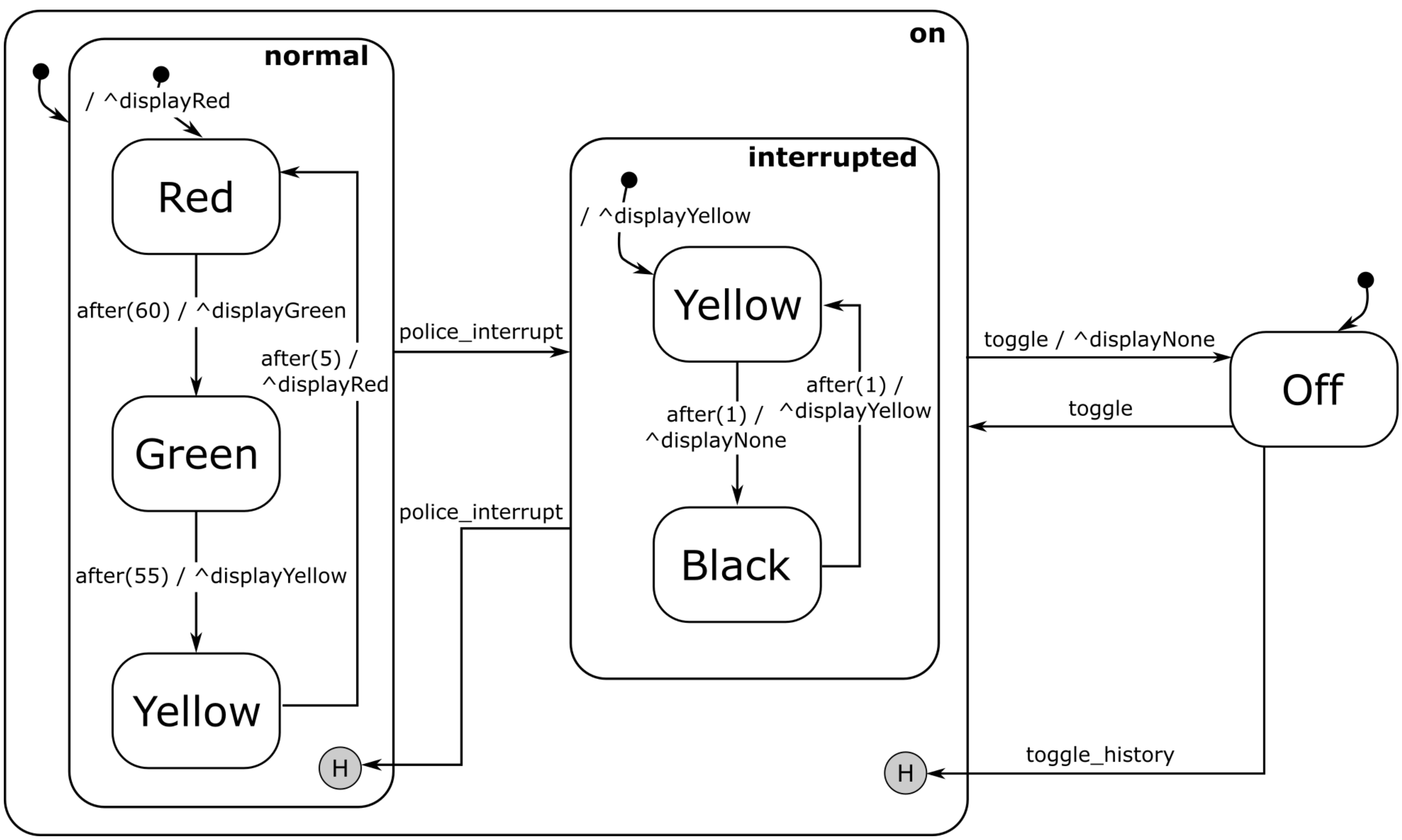


Add another hierarchy level that supports switching on and off the coomplete traffic light. Go into detail with shallow and deep histories.

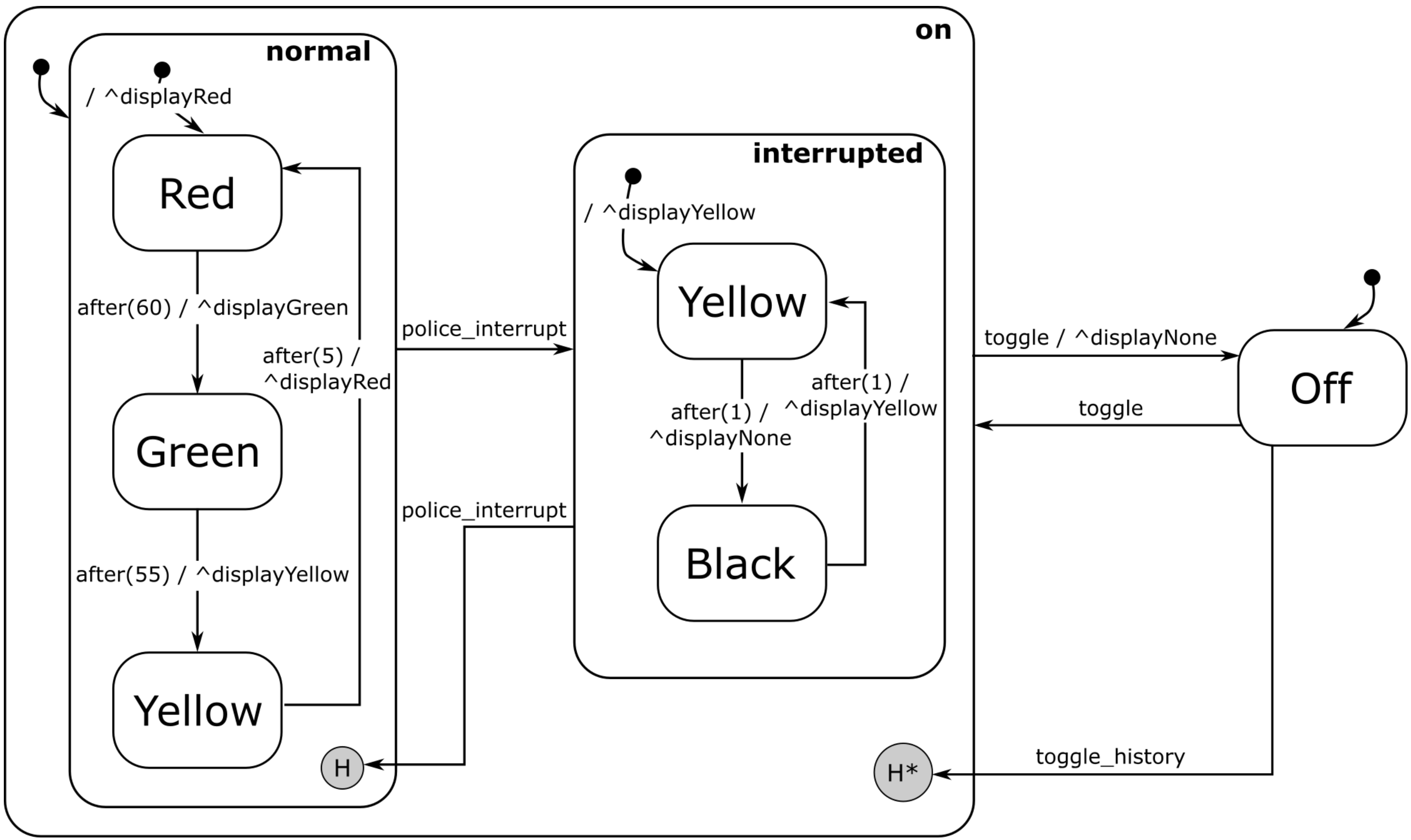
- R9: The traffic light can be switched on and off.
- R9a: The traffic light is initially off.
- R9b: If the traffic light is off nocht light is on.
- R9c: After switching off and on again the traffic light must switch on the previously activated light.



Exercise 7: Solution



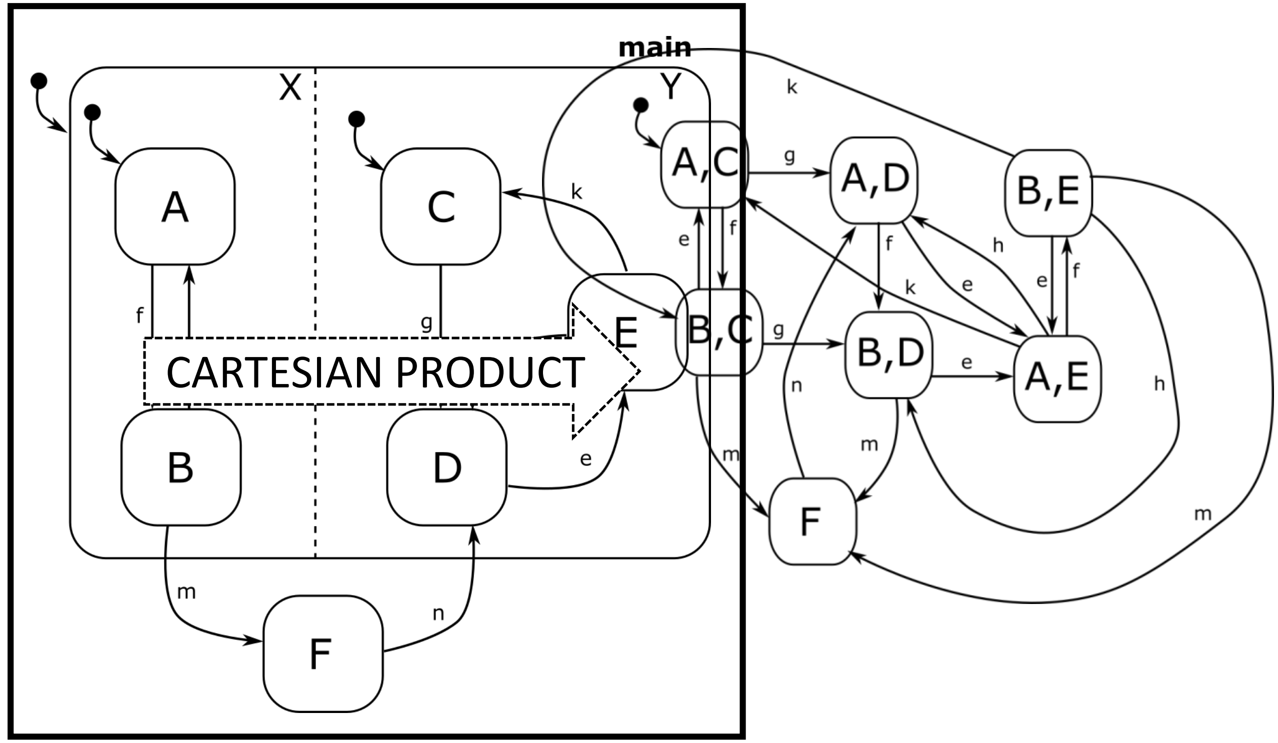
Exercise 7: Alternative Solution



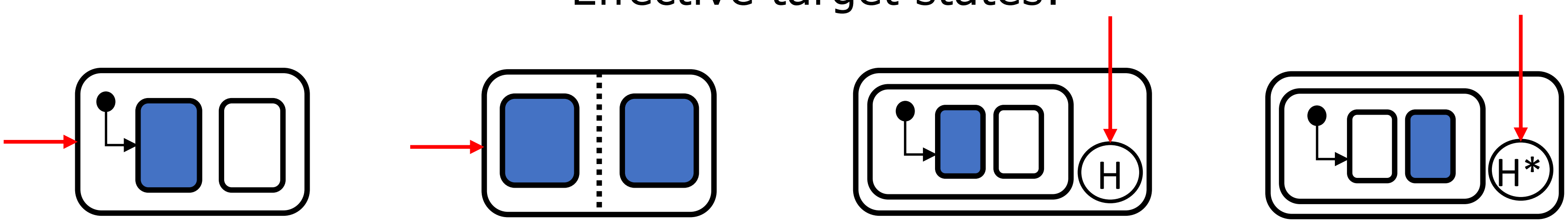
Orthogonality

Orthogonality

Semantics/Meaning?

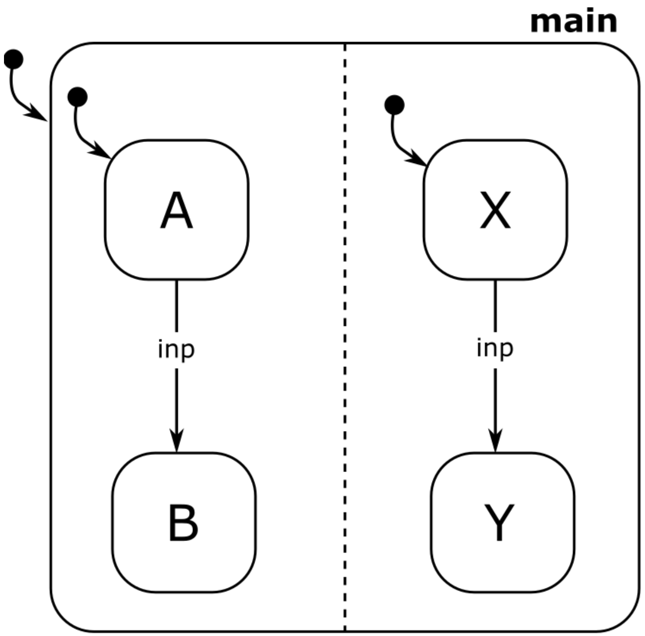


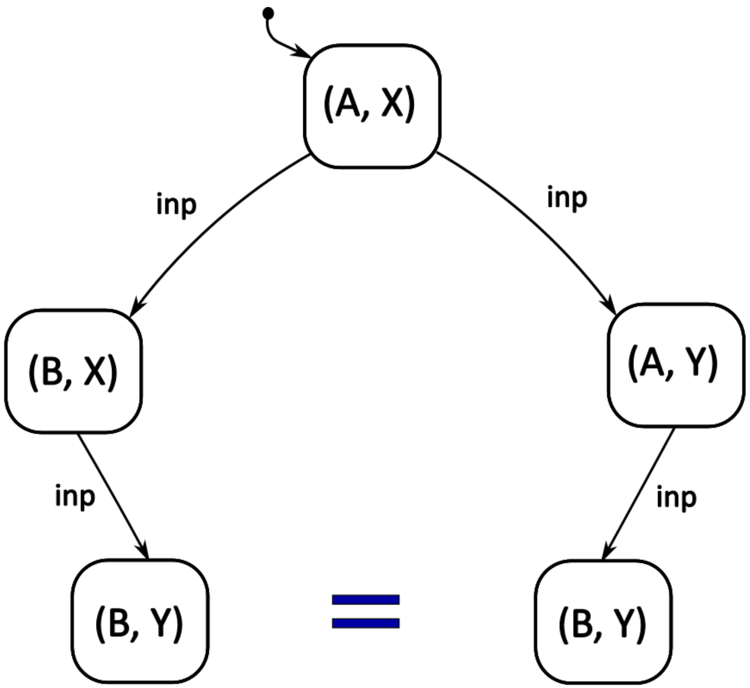
Effective target states:



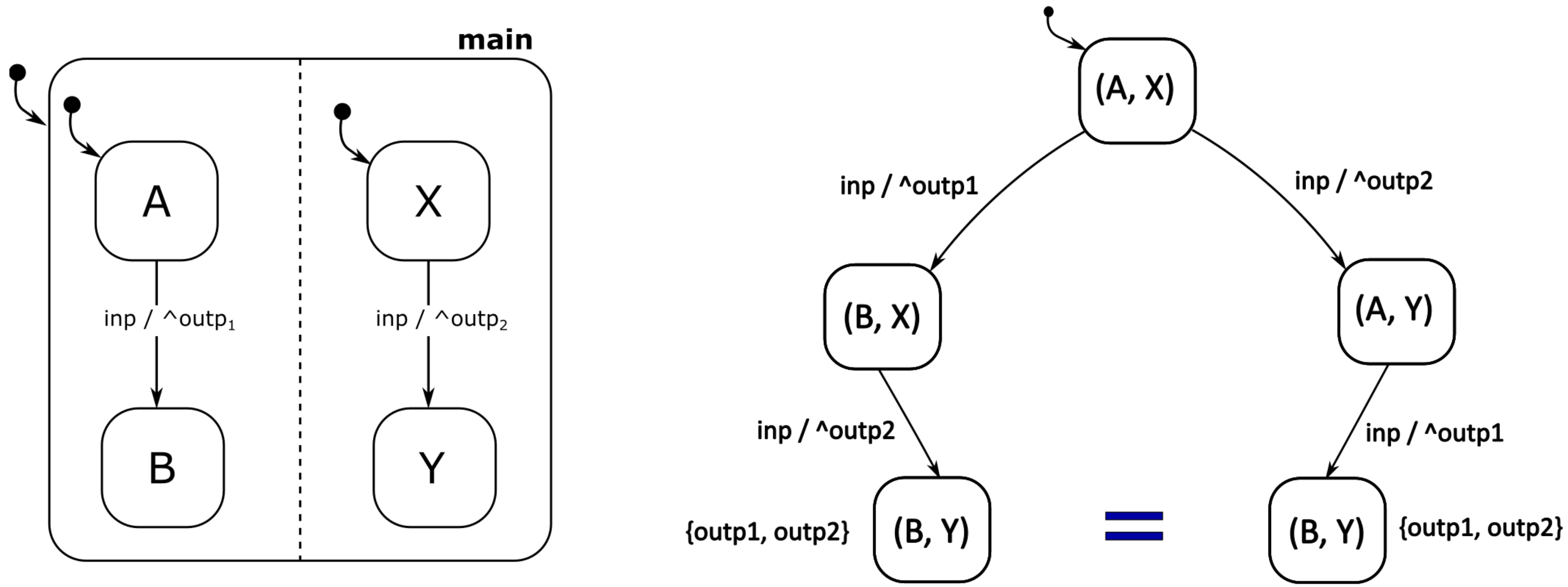
RECURSIVE!

Parallel (In)Dependence

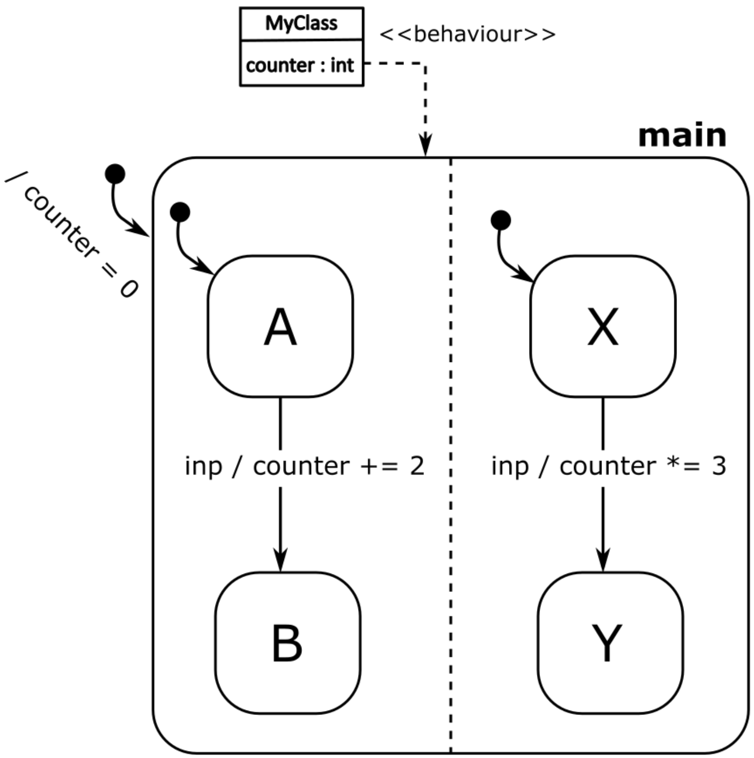


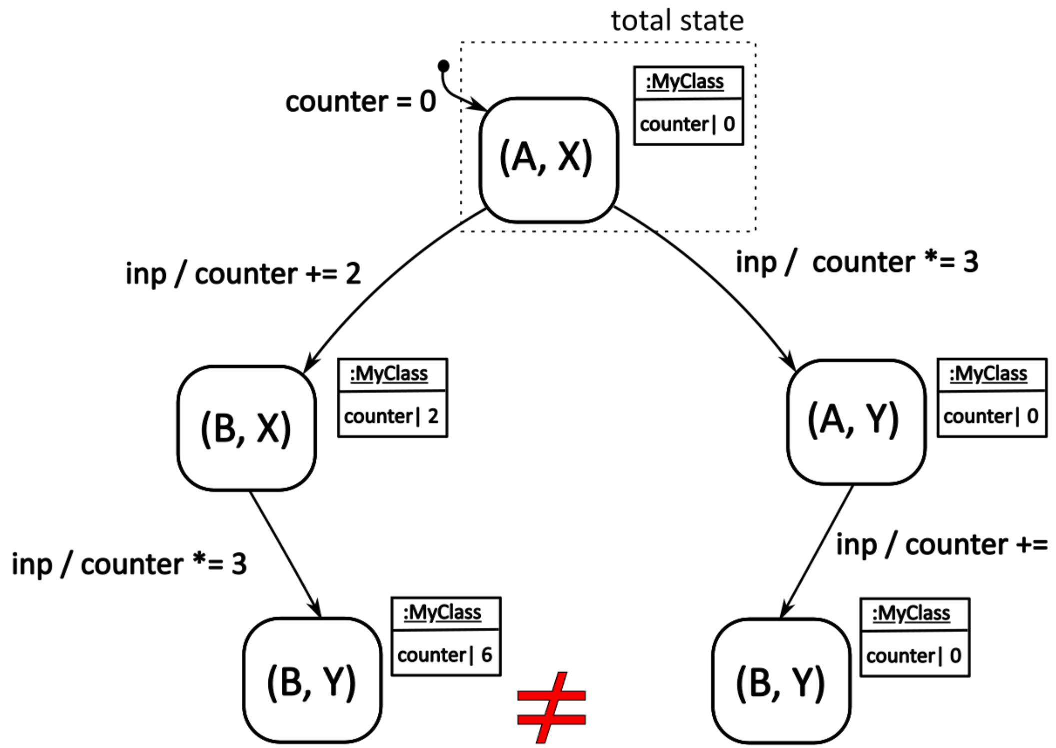


Parallel (In)Dependence

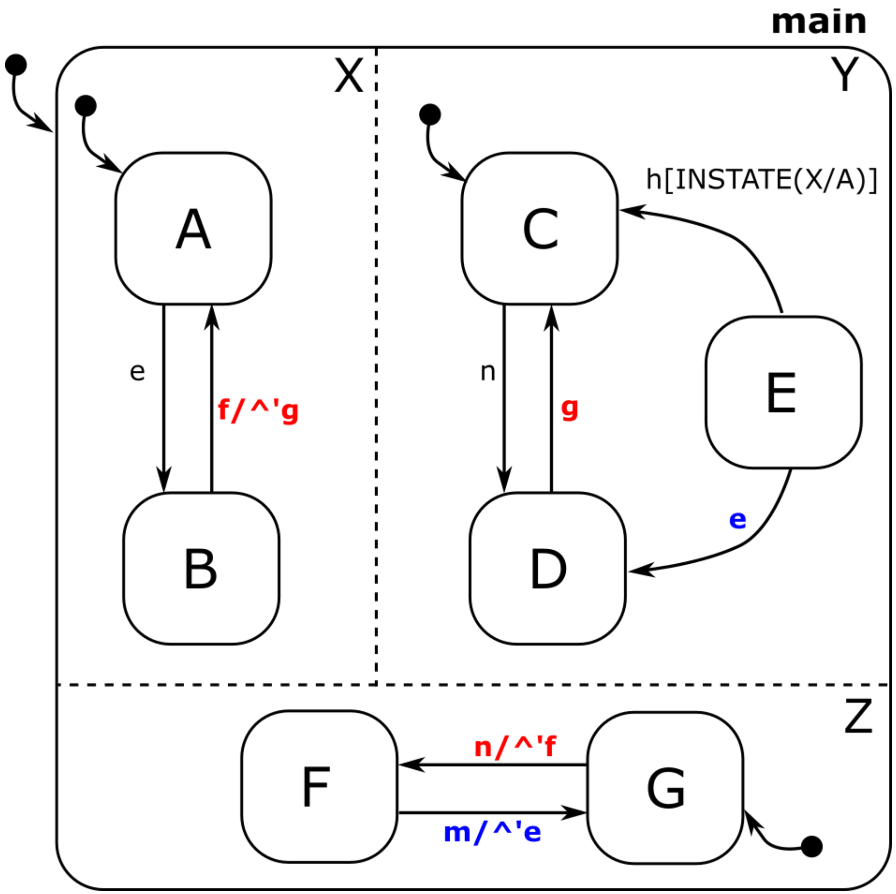


Parallel (In)Dependence





Orthogonality: Communication



Input Segment: **nmnn**

- Components can communicate:
  - raising local events:

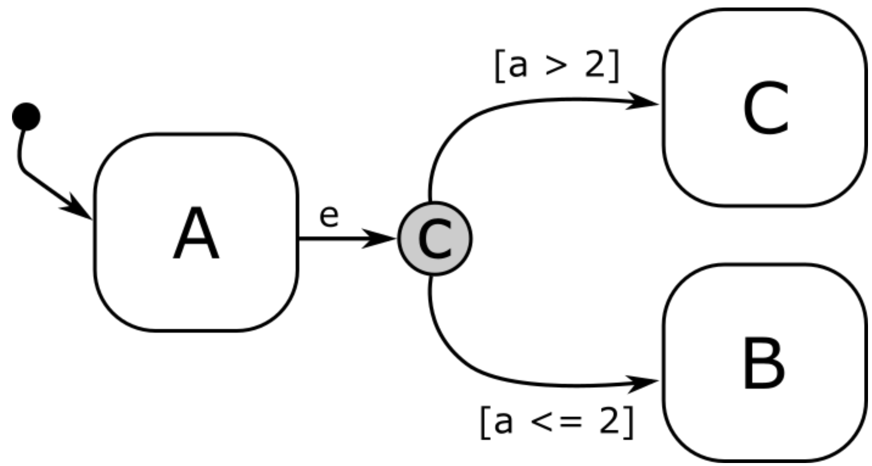
    ^’<<event name>>
  - INSTATE macro

    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
- Always an “else” branch required!
- Equivalent (in this case) to two transitions:
  - A – *e[a > 2]* -> C
  - A – *e[a <= 2]* -> B

Exercise 8

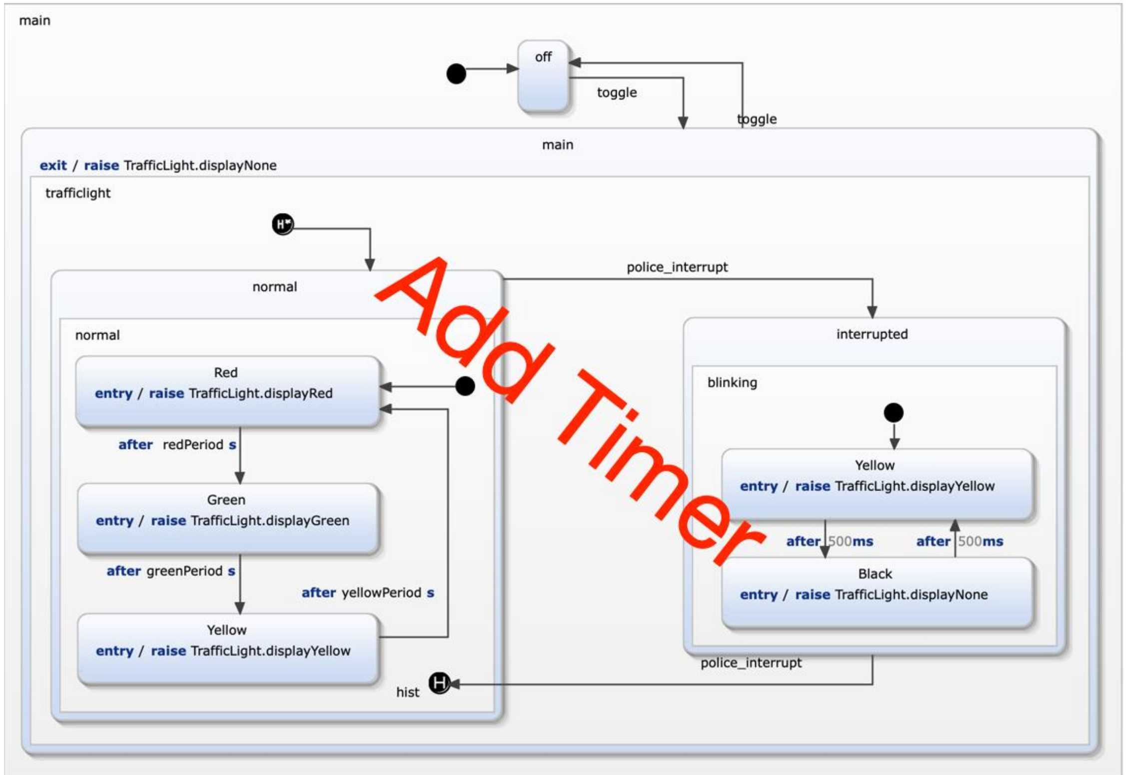
Add a timer to the traffic light

Exercise 8: Requirements

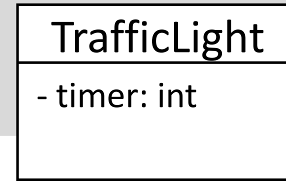
TrafficLight
- timer: int

In this exercise a timer must be modeled. It introduces using orthogonal regions.

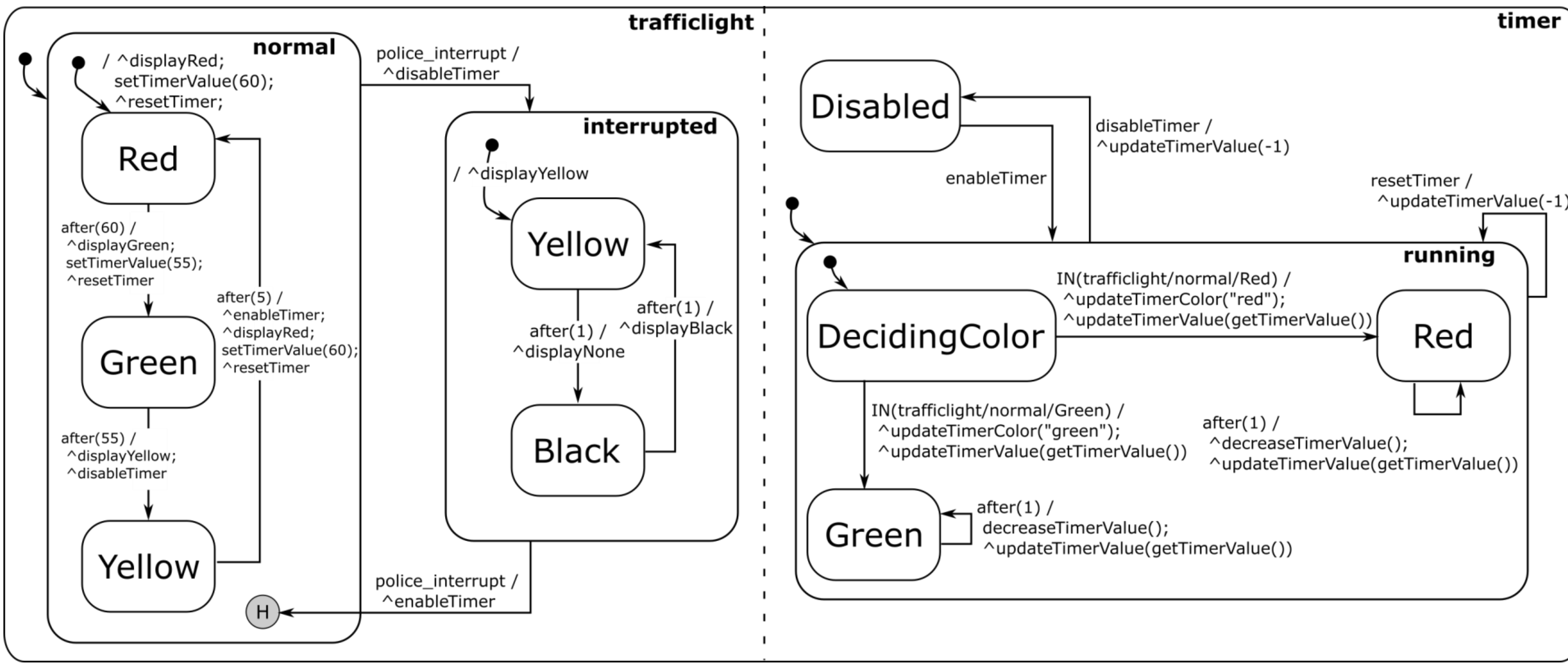
- R10a: A timer displays the remaining time while the light is red or green
- R10b: This timer decreases and displays its value every second.
- R10c: The colour of the timer reflects the colour of the traffic light.



Exercise 8: Solution

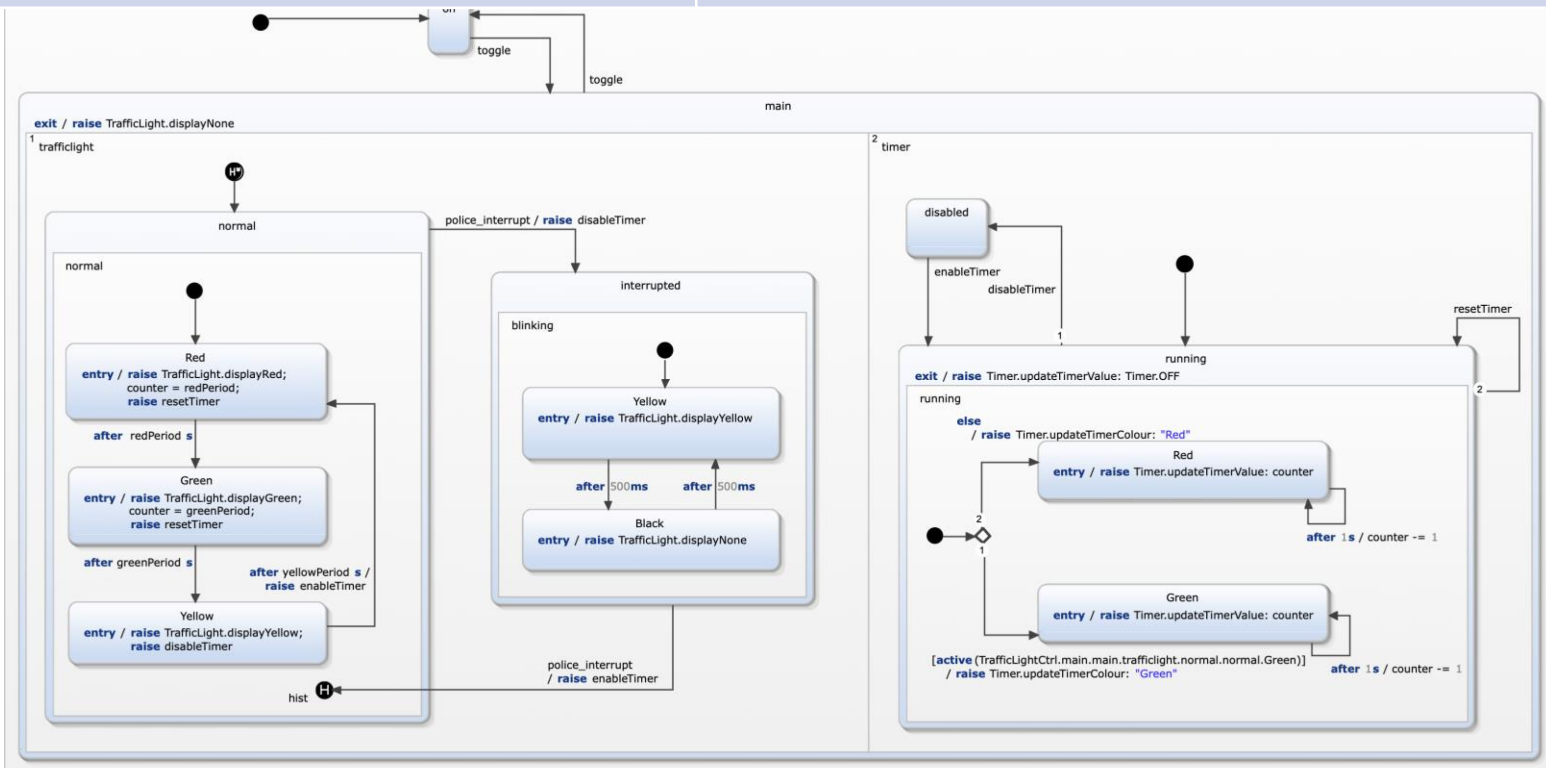


- R10a: A timer displays the remaining time while the light is red or green
- R10b: This timer decreases and displays its value every second.
- R10c: The colour of the timer reflects the colour of the traffic light.



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.
R10a: This timer decreases and displays its value every second.	An internal variable for the counter is introduced. Wehn switching e traffic light phase the counter value is set to the time period of the phase. Additionally the local evemts 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 state using active() function.



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 requirements
  - ROM and RAM memory
  - Debug capabilities
  - Clarity and maintainability

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 several frameworks like Spring Statemachine, Boost MSM or Qt State Machine Framework
- Each State becomes one class
- All classes derive from a common interface

```java
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";
    }

}
```

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**

TrafficLightCtrl.sct | TrafficLightCtrlStatemachine.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() {
```

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
```

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 =  ""
        }
    }
}
```

Setup Code (Excerpt)

```
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();
}
```

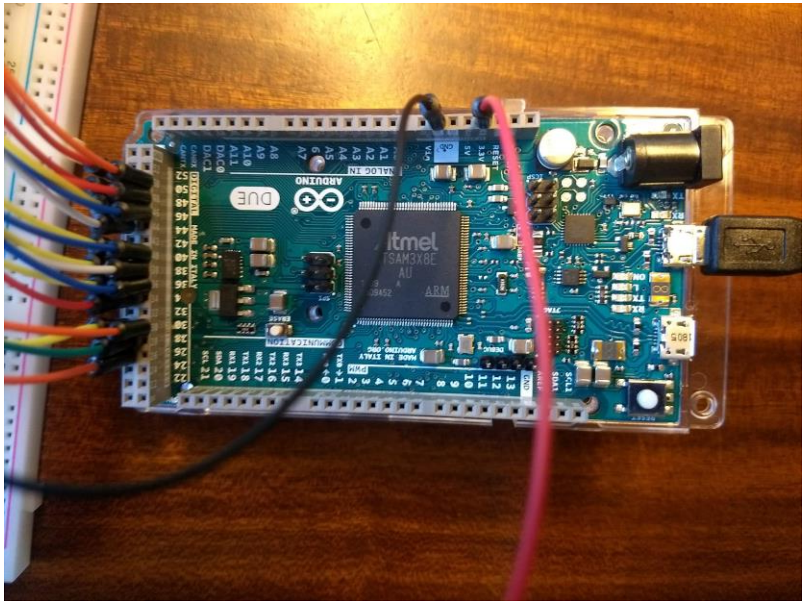
Runner

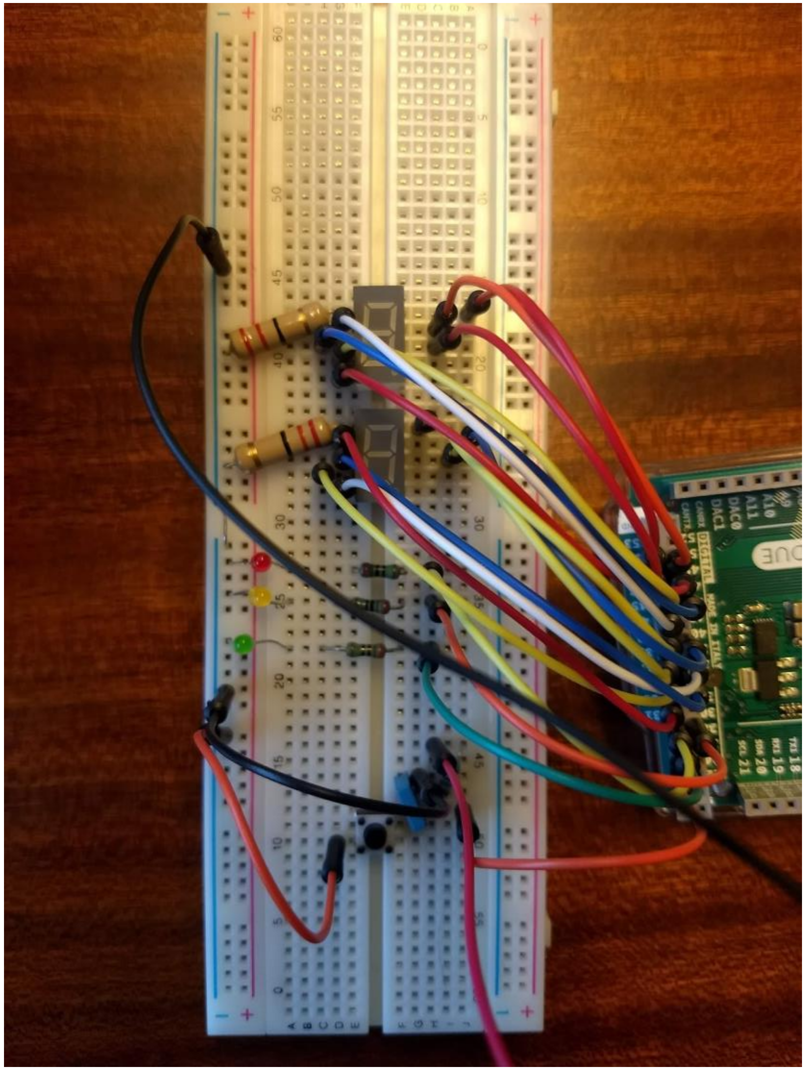
```
protected void run() {
    statemachine.enter();
    RuntimeService.getInstance().registerStatemachine(statemachine, 100);
}
```

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_cour
  }
}
```

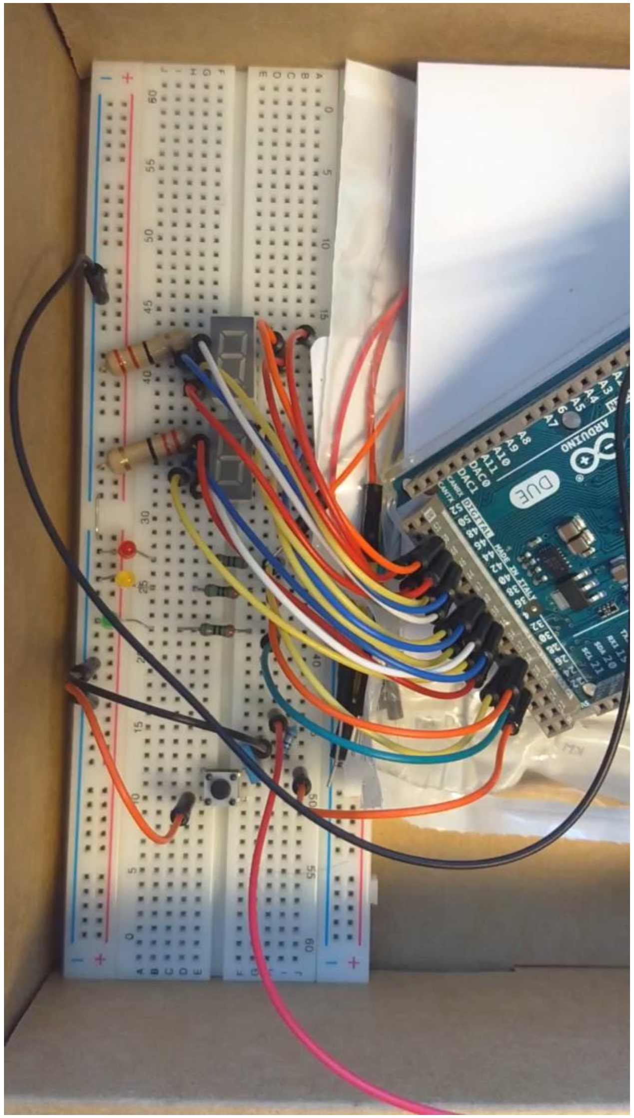
Generator

```
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;
}
```

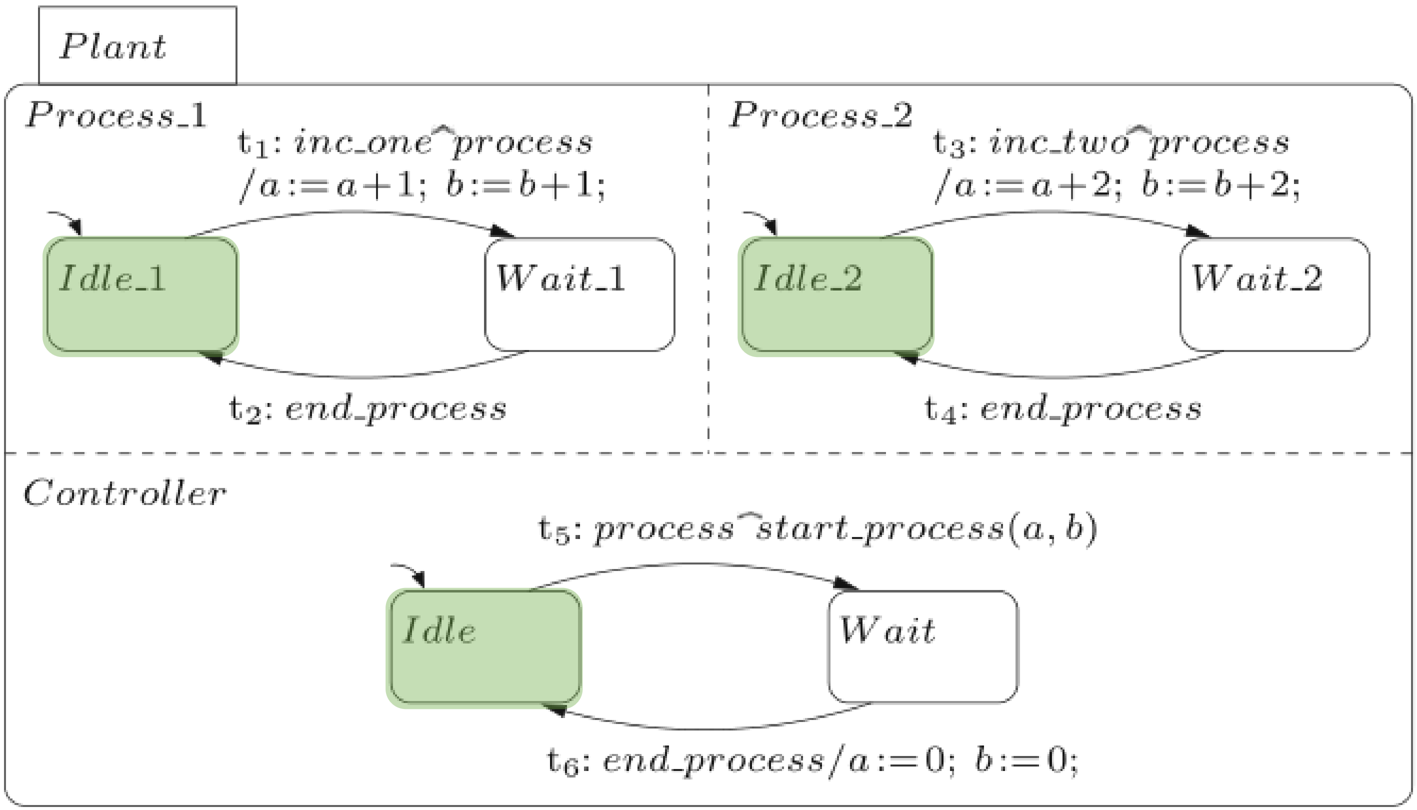
Deployed Application



Semantic Choices

Semantic Choices

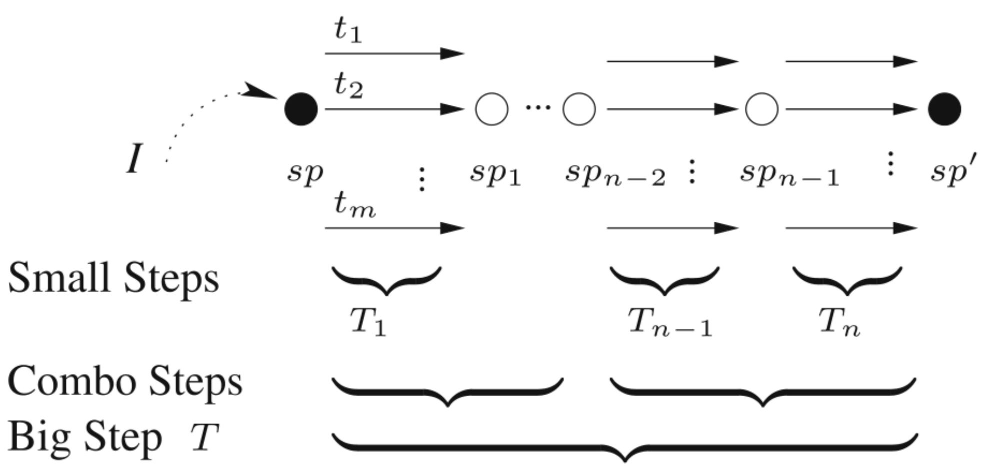
enabled events: [*inc_one*, *inc_two*]



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

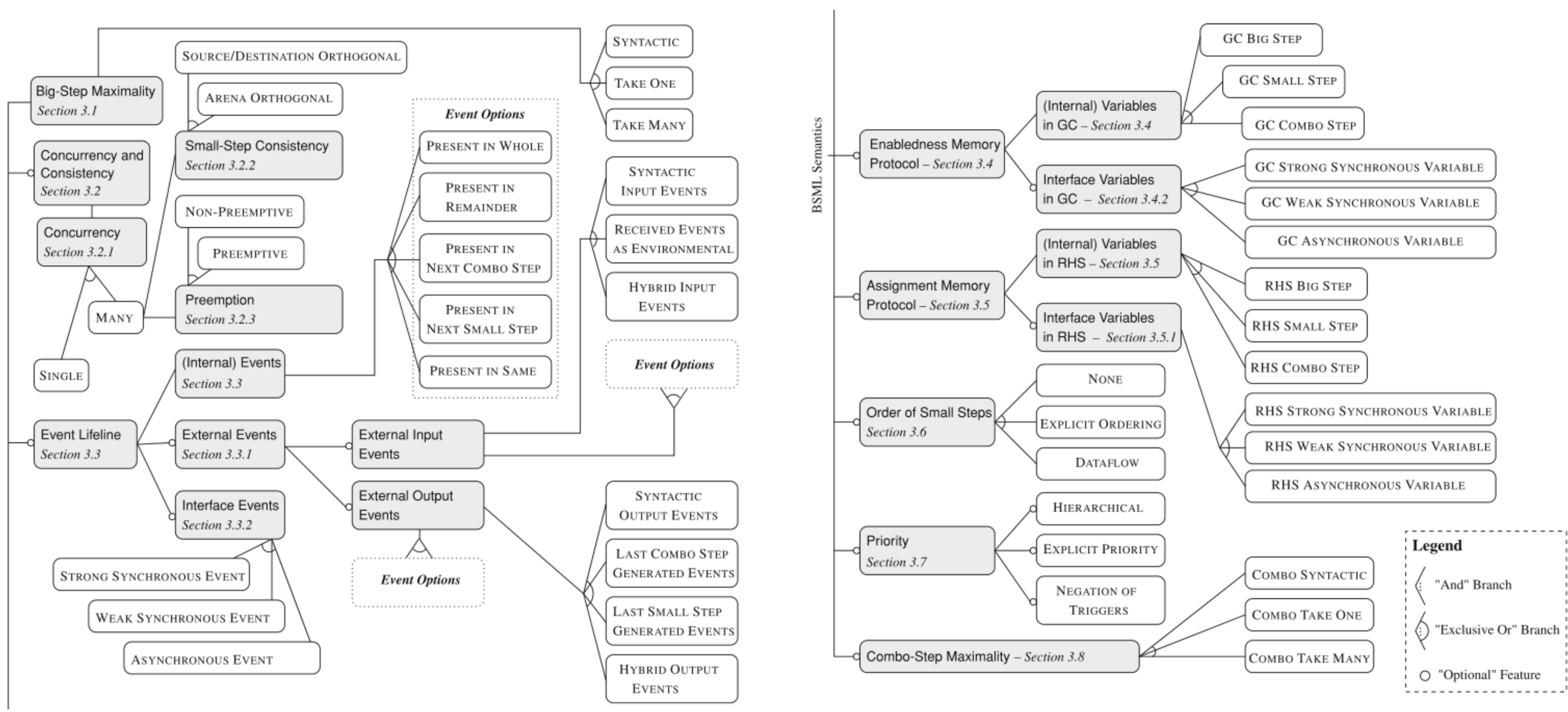
Big Step, Small Step

- 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.

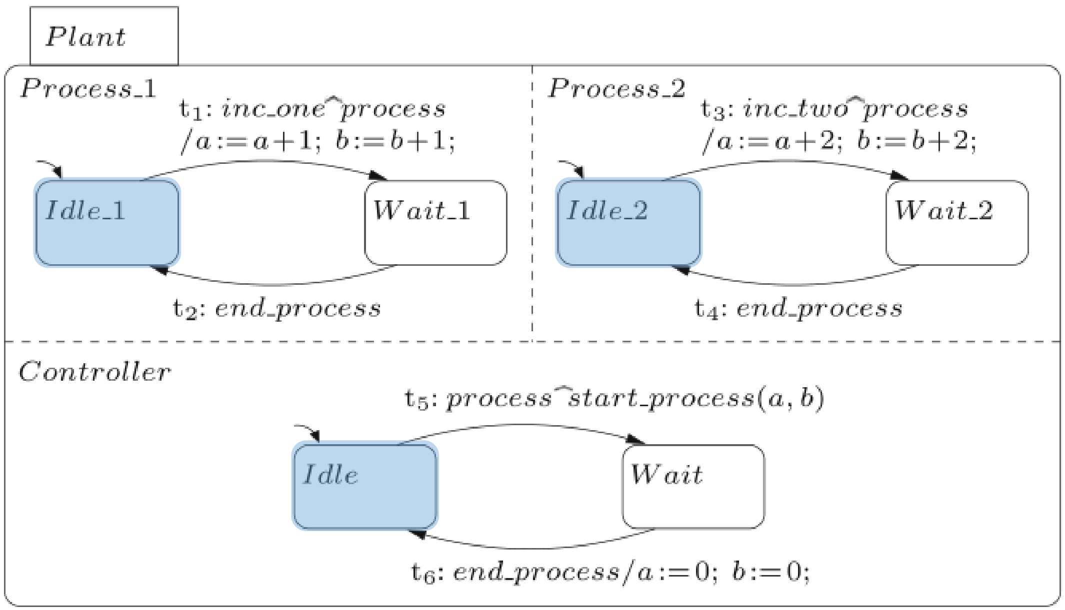
Semantic Options



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

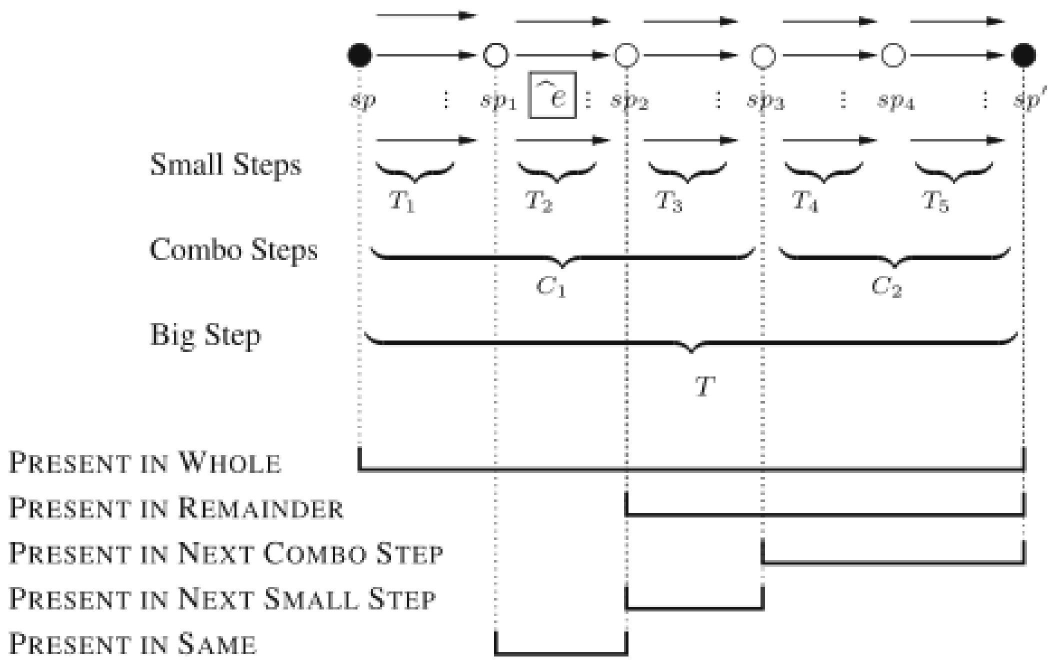
<{t3}, {t1}, {t5}>

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



Semantic Options: Examples

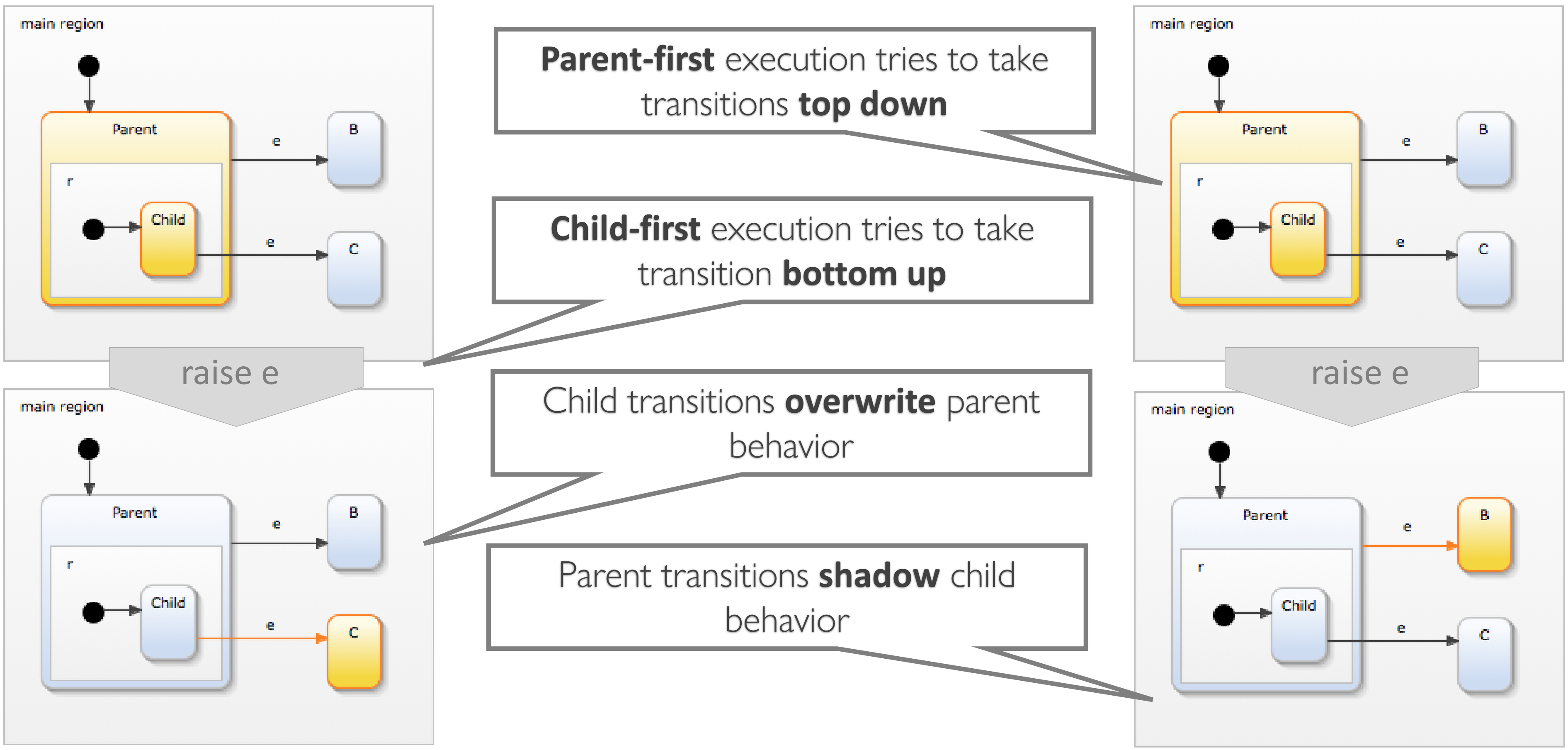
	Rhapsody	Statemate	(Default) SCCD
Big Step Maximality	Take Many	Take Many	Take Many
Internal Event Lifeline	Queue	Next Combo Step	Queue
Input Event Lifeline	First Combo Step	First Combo Step	First Combo Step
Priority	Source-Child	Source-Parent	Source-Parent
Concurrency	Single	Single	Single

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.

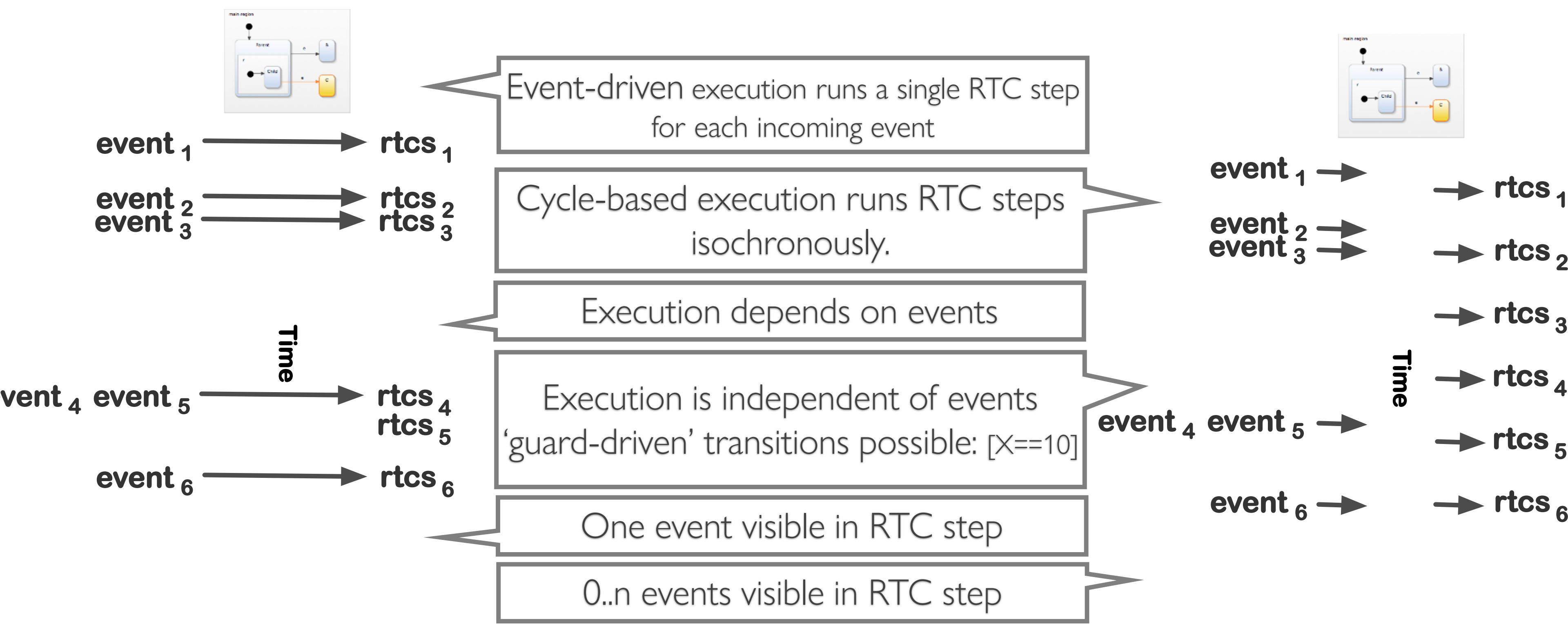
Child-first vs Parent-first
Event-driven vs Cycle-based

Composite States Execution

In which order are transitions evaluated in a HSM?

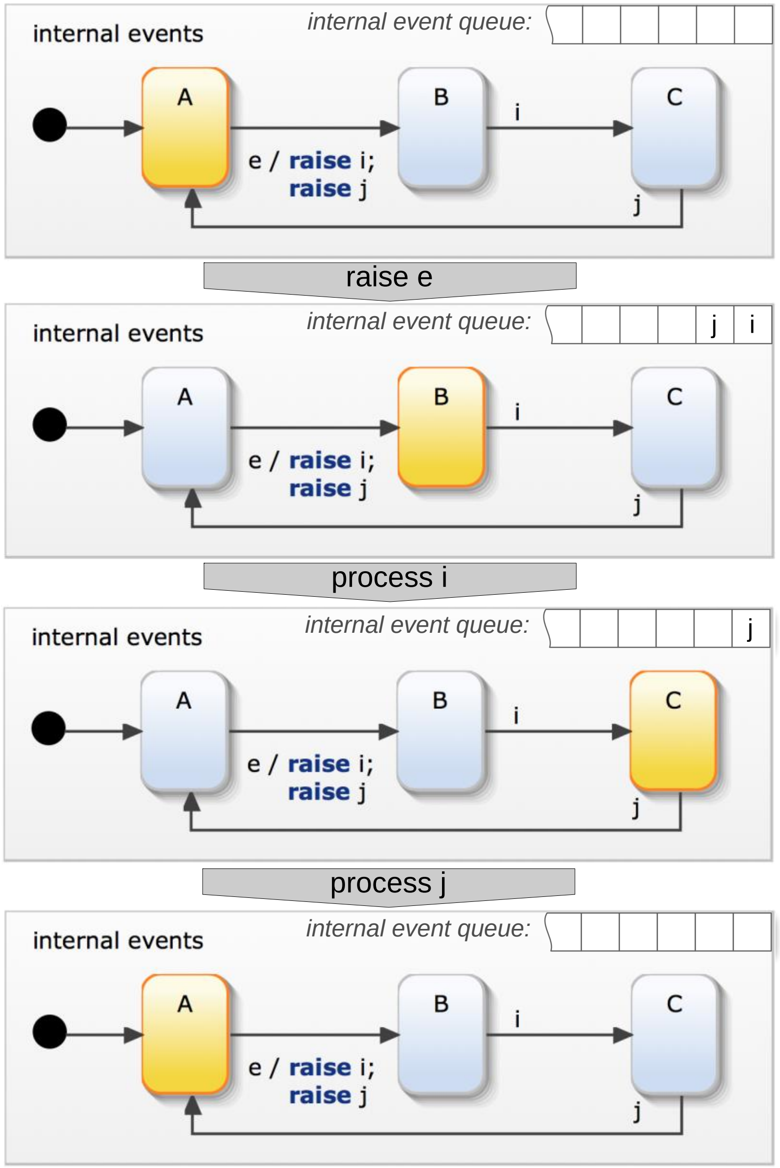


The behavior of state machines are executed in single 'run-to-completion' steps.



Event-driven: event queuing

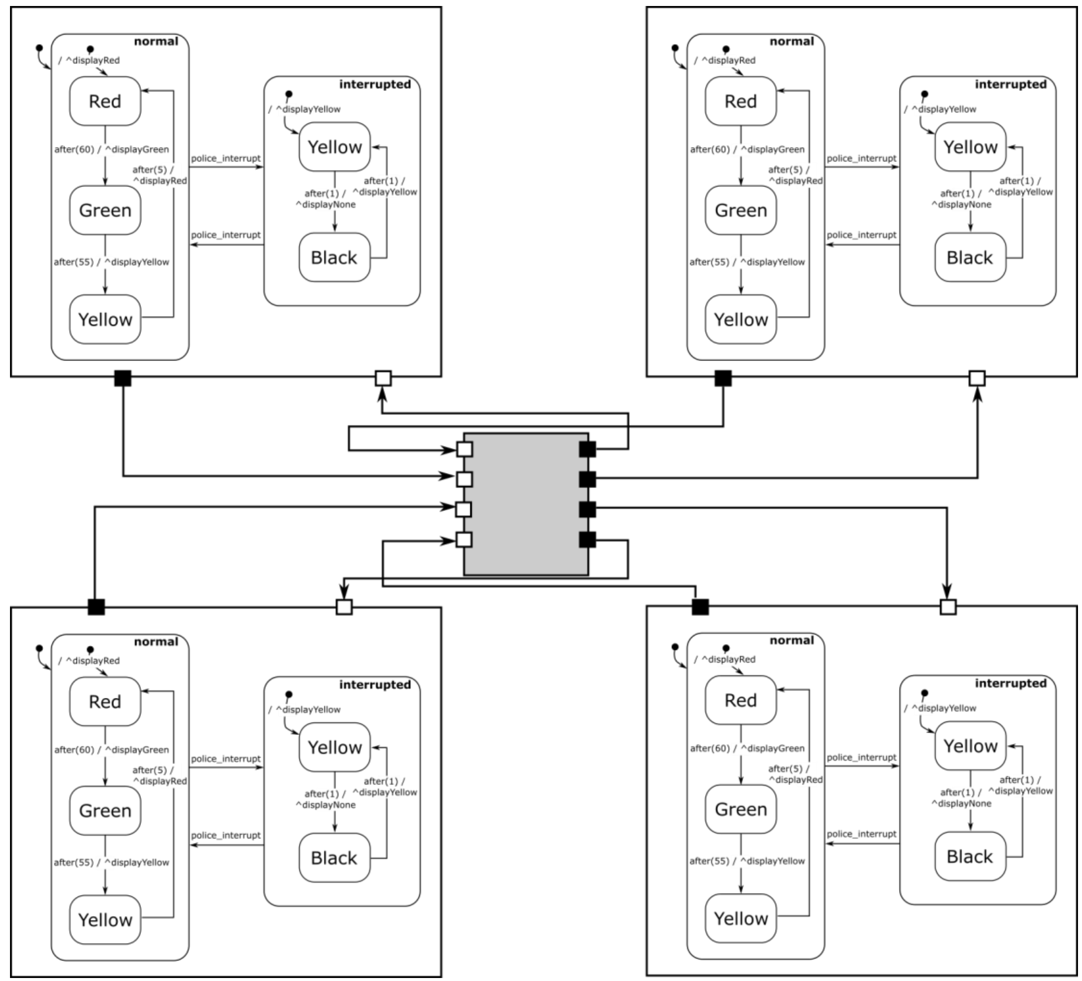
- multi-step RTC in event-driven execution
- all internal & in events raised within a RTC are processed
- each in event is processed by a single step which are composed to a RTC step
- makes use of event queues: in & internal
- internal event have higher priority than in events



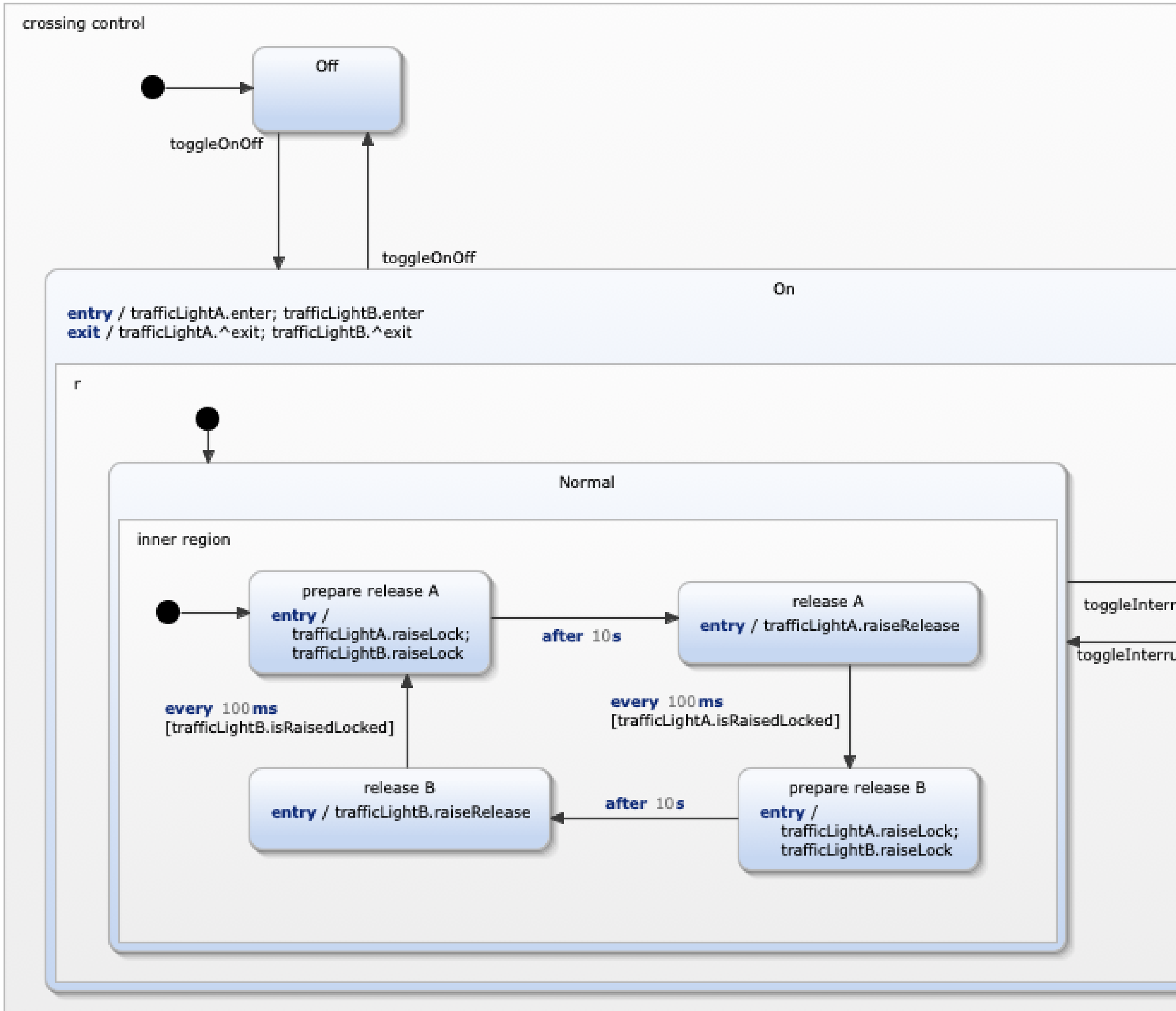
Composition

Composition of Statecharts

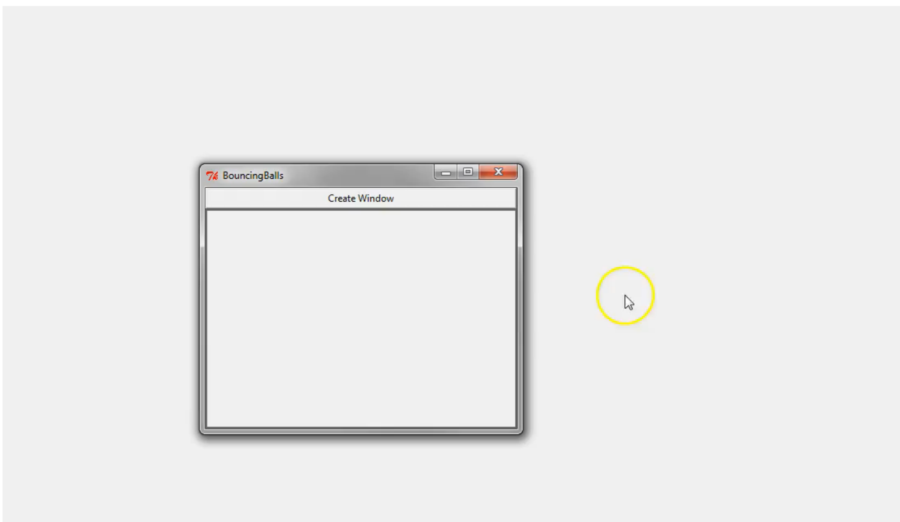
- Composition of multiple Statechart models
  - Instantiation
  - Communication
  - Semantics
- Often solved in code…



Composition Example



Dynamic Structure: SCCD



Behavior

- Timed
- Autonomous
- Interactive
- Hierarchical

Structure

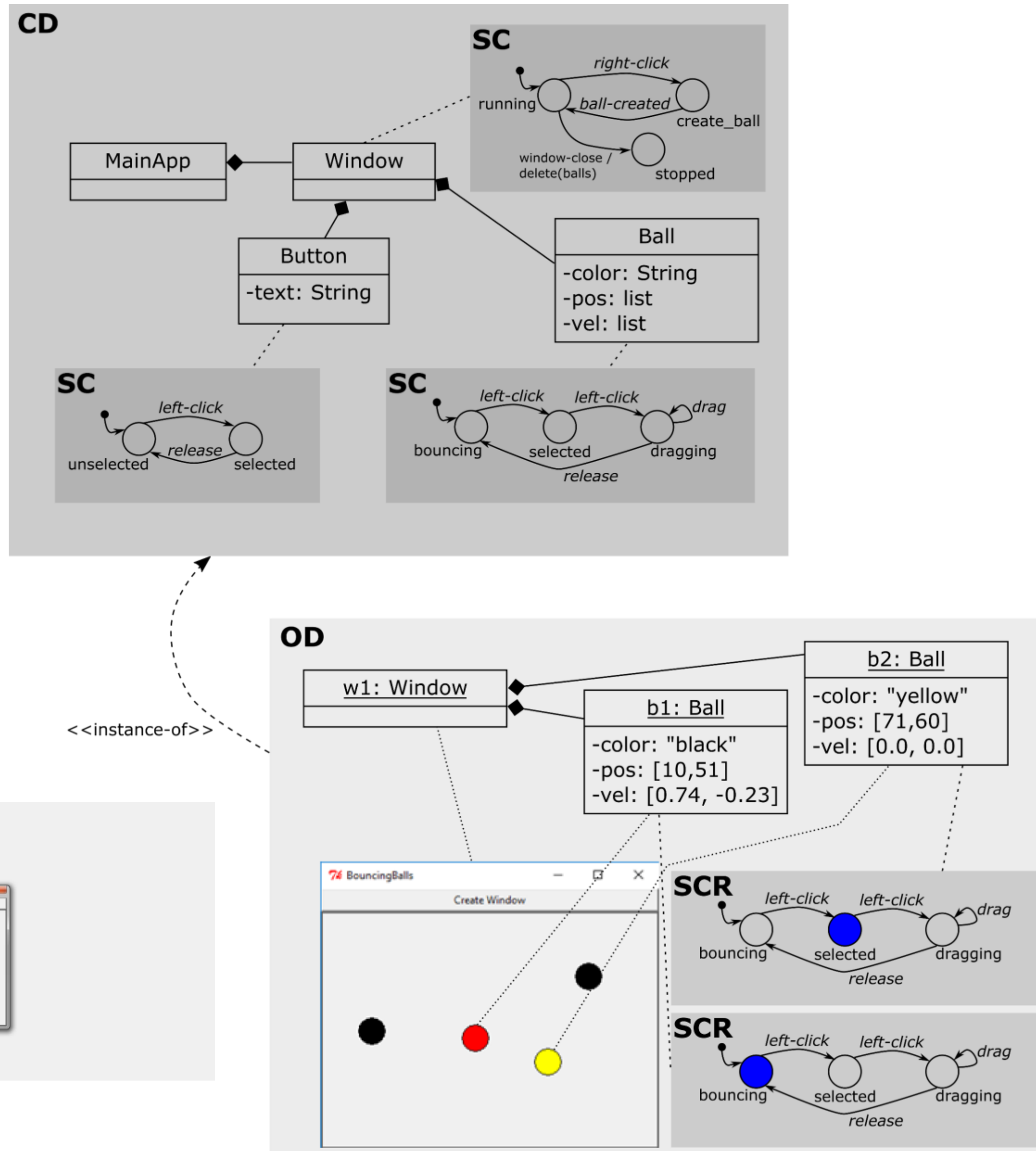
- Dynamic
- Hierarchical

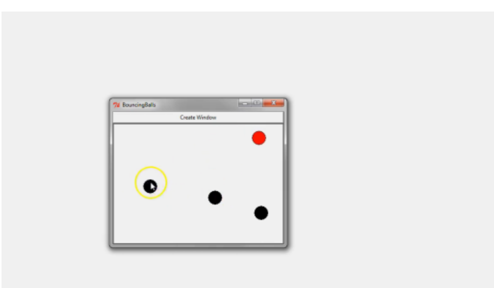
Design? Statecharts + ???

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

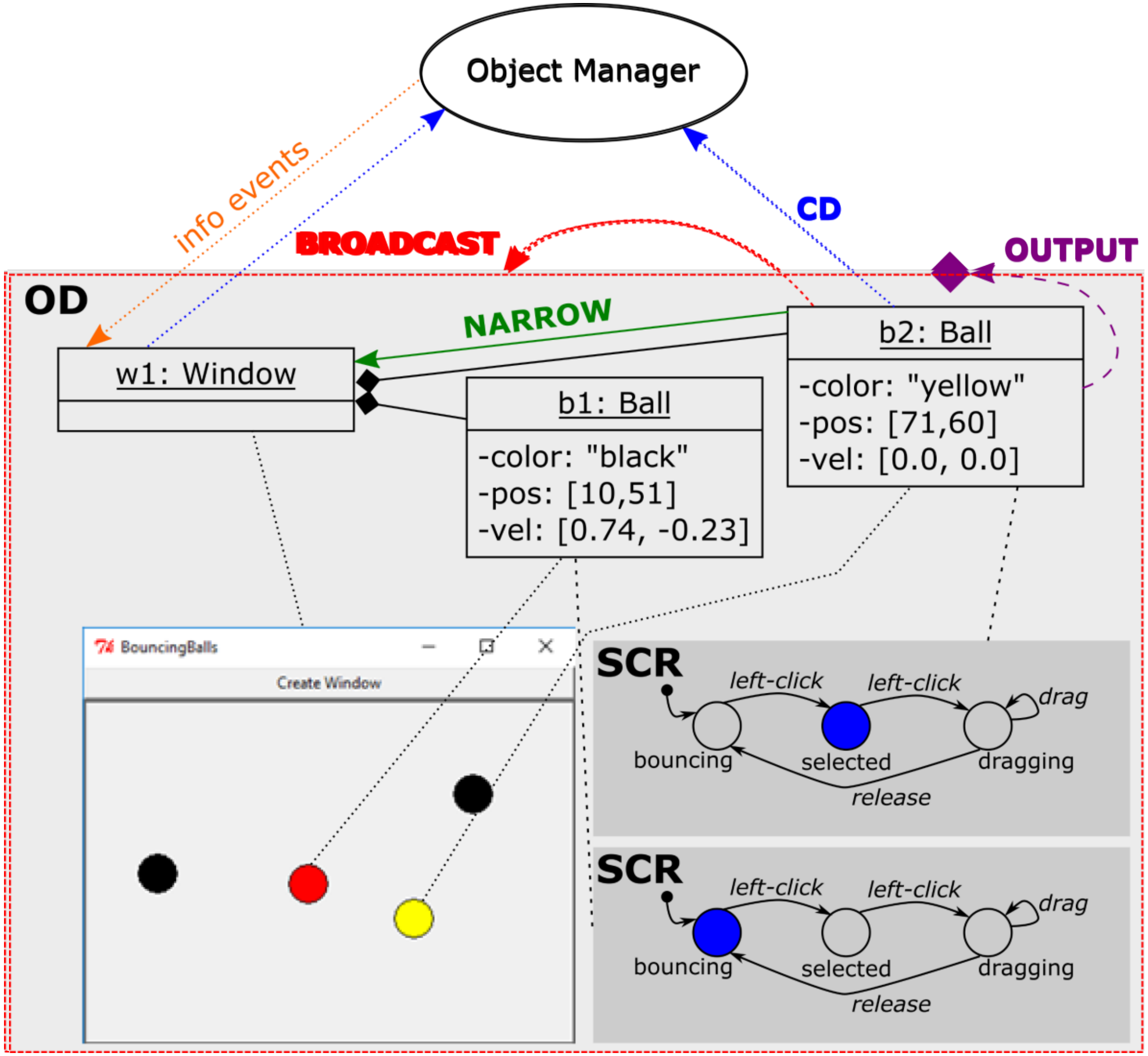
Simon Van Mierlo, Yentl Van Tendeloo, Bart Meyers, Joeri Exelmans, and Hans Vangheluwe. SCCD: SCXML extended with class diagrams. In 3rd Workshop on Engineering Interactive Systems with SCXML, part of EICS 2016, 2016.

SCCD: Conformance

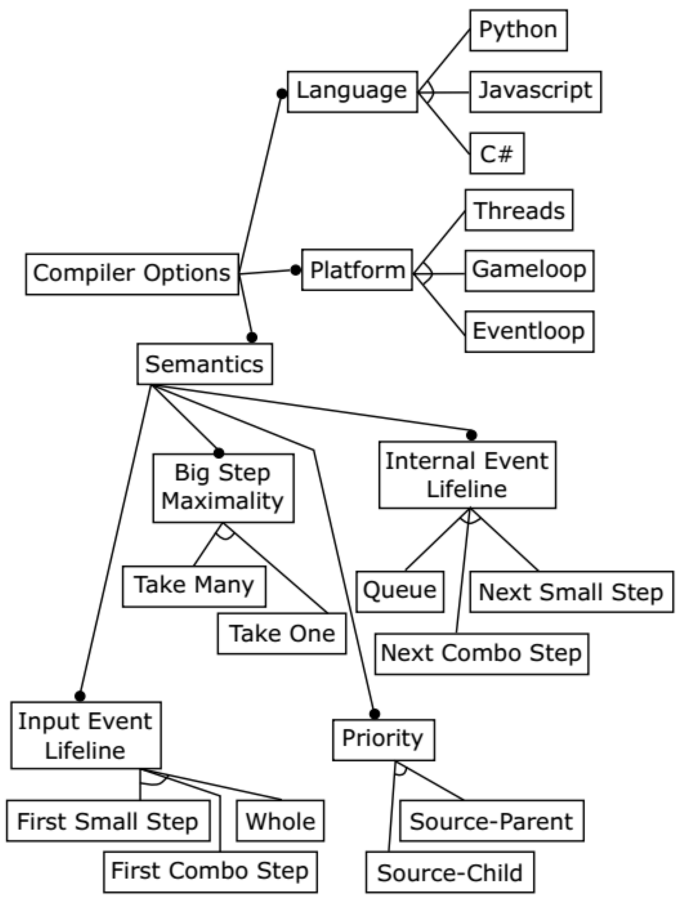


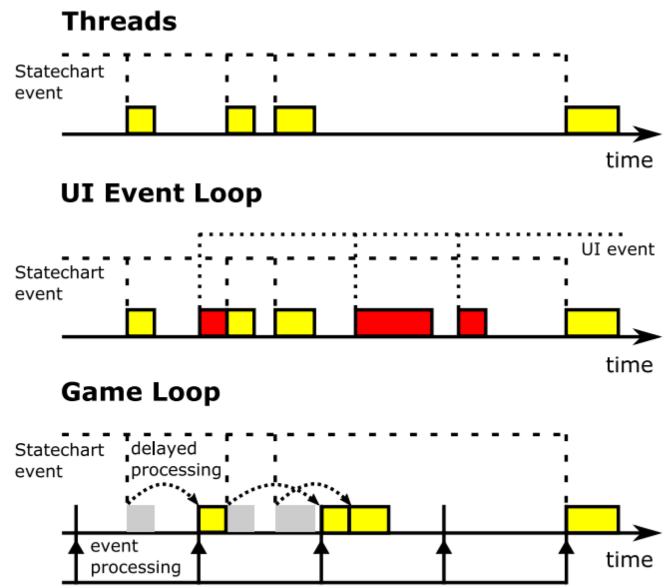


Communication: Event Scopes



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