Developing Reactive Systems Using Statecharts

Simon Van Mierlo  
University of Antwerp - 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
<table>
<thead>
<tr>
<th>Time</th>
<th>Activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>09:00 – 10:30</td>
<td>Introduction, Yakindu set-up</td>
</tr>
<tr>
<td>10:30 – 11:00</td>
<td>Coffee Break</td>
</tr>
<tr>
<td>11:00 – 12:30</td>
<td>Tutorial: Statecharts Concepts + Exercises</td>
</tr>
<tr>
<td>12:30 – 14:00</td>
<td>Lunch</td>
</tr>
<tr>
<td>14:00 – 15:30</td>
<td>Tutorial: Statecharts Concepts + Exercises</td>
</tr>
<tr>
<td>15:30 – 16:00</td>
<td>Coffee Break</td>
</tr>
<tr>
<td>16:00 – 17:30</td>
<td>Tutorial: Advanced Concepts</td>
</tr>
</tbody>
</table>
Introduction
Reactive Systems

- Complexity:
  - Reactive (to events), timed, concurrent, behaviour

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)?


"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”
Discrete-Event Abstraction

behavioural model

moving

W

key_up / ^move_up
key_down / ^move_down

U

D

low_fuel

[low_fuel_detected()] / ^low_fuel

[fuel_ok()]

after(3s)

shooting

key_enter / ^shoot

TANK WARS
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

<table>
<thead>
<tr>
<th>Event</th>
<th>State</th>
<th>State</th>
<th>State</th>
<th>State</th>
<th>State</th>
<th>State</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>s₁, n</td>
<td>s₂, n</td>
<td>s₃, n</td>
<td>s₄, n</td>
<td>s₅, n</td>
<td>s₆, n</td>
<td>s₆, 5</td>
</tr>
<tr>
<td>10</td>
<td>s₂, n</td>
<td>s₃, n</td>
<td>s₄, n</td>
<td>s₅, n</td>
<td>s₆, n</td>
<td>s₆, 5</td>
<td>s₆, 10</td>
</tr>
<tr>
<td>25</td>
<td>s₅, n</td>
<td>s₆, n</td>
<td>s₆, 5</td>
<td>s₆, 10</td>
<td>s₆, 15</td>
<td>s₆, 20</td>
<td>s₆, 25</td>
</tr>
<tr>
<td>O</td>
<td>s₀, n</td>
<td>s₁, n</td>
<td>s₂, n</td>
<td>s₃, n</td>
<td>s₄, n</td>
<td>s₅, n</td>
<td>s₀, orange juice</td>
</tr>
<tr>
<td>R</td>
<td>s₀, n</td>
<td>s₁, n</td>
<td>s₂, n</td>
<td>s₃, n</td>
<td>s₄, n</td>
<td>s₅, n</td>
<td>s₀, apple juice</td>
</tr>
</tbody>
</table>
Mealy and Moore Machines

Moore Machines
- Output only depends on current state. \( \lambda: Q \rightarrow O \)
- Input: 00 -> Output: 111

Mealy Machines
- Output depends current state and current input. \( \lambda: Q \times \Sigma \rightarrow O \)
- 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

**Higraphs**

**Euler Diagrams**

All $A$ are $B$. No $A$ is $B$. Some $A$ is in $B$. Some $A$ is not in $B.$

- topological notions for set union, difference, intersection

**Hypergraphs**

*Unordered Cartesian Product*

$A = B \times C$

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

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

Higraphs

Euler Diagrams

Hypergraphs

Unordered Cartesian Product

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

Statecharts History

- Incorporated in UML: State Machines (1995)
  https://www.w3.org/TR/scxml/
- Standard: Precise Semantics for State Machines (2019)  
  https://www.omg.org/spec/PSSM/
STATEMATE: A Working Environment for the Development of Complex Reactive Systems

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

https://ptolemy.berkeley.edu/ptolemyII/ptll11.0/index.htm


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


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

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

Controller

<table>
<thead>
<tr>
<th>plant input</th>
<th>plant output</th>
</tr>
</thead>
</table>

(Physical) Plant

123

<<sense>>

<<act>>

System

<<sense>>

<<sense>>

Environment

Controller

(Physical) Plant

System

Environment
What are we developing?

- Turn on/off traffic lights (red/green/yellow)
- Display counter value (three-digit)
- Change counter colour (red/green)
- Sense button presses
- Autonomous (timed) behaviour
- Interrupt logic
- Orthogonal (traffic light/timer) behaviour

(Deployed) Statecharts Model

"Interface"

System

Controller

(Physical) Plant

Environment

<<sense>>

<<act>>
Deployment (Simulation)

Controller

(Simulated) Plant

System

Environment

1

2

plant input

plant output

<<sense>>

<<sense>>

<<act>>

Environment

System

Controller

(Simulated) Plant

plant input

plant output

<<sense>>

<<act>>
Deployment (Hardware)

Controller

(Physical) Plant

System

Environment

Controller

plant input

plant output

<<sense>>

<<sense>>

<<act>>

Environmental

Sensor

Actuator

Controller

plant input

plant output

<<sense>>

<<act>>
Workflow


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 …
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
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”
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

- **Spontaneous**
  
  ![Spontaneous Transition Diagram]

- **Input Event**
  
  ![Input Event Transition Diagram]

- **After Event**
  
  ![After Event Transition Diagram]

```
event(params) / output_action(params)
```

```
event queue
\[ \ldots \ e_3 \ e_1 \ e_5 \]\n```

```
event queue
\[ e_3 \ e_2 \]\n```

```
\[ \ldots \ t_3 \ t_2 \]\n```

Transitions: Raising Output Events

Syntax for output action:

\[^\text{output\_event}\]

means “raise the event output\_event (to the environment)”
The YAKINDU Statechart Dialect
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: \( \text{trigger} [ \text{guard} ] / \text{effect} \)

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

- 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

Environment (Simulated) Plant

<<observe>>
### Exercise 1 - Solution

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

<table>
<thead>
<tr>
<th>DataStore</th>
</tr>
</thead>
<tbody>
<tr>
<td>- var₁: t₁ = val₁</td>
</tr>
<tr>
<td>- var₂: t₂ = val₂</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>- varₙ: tₙ = valₙ</td>
</tr>
</tbody>
</table>
Full System State: Initialization

**Initial state**
- exactly one per model
- “entry point”

**Provide default value**
- for each variable
- “initial snapshot”

**DataStore**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>- var_1: t_1 = val_1</td>
<td></td>
</tr>
<tr>
<td>- var_2: t_2 = val_2</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>- var_n: t_n = val_n</td>
<td></td>
</tr>
</tbody>
</table>

**Compare:**
- C++ initialization
- *implicit* state
  - (program counter)
  - + data store

```cpp
1 int main() {
2   ...
3 }
```
Transitions: Guards

- Modelled by action code in some appropriate language

- Spontaneous

- Data Store Variable

- Parameter Variable
Transitions: Output Actions

- **Output Event**
  \[ ^\text{output\_event}(p_1, p_2, \ldots, p_n) \]
- **Assignment** (to the non-modal part of the state)
  - by action code in some appropriate language

\[ \text{event}(\text{params}) \ [\text{guard}] \ / \ \text{output\_action}(\text{params}) \]
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
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
In statecharts variables hold quantitative values.

Variables may be accessible from 'outside' the statechart

Variables behave like you would expect

```plaintext
// 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

```plaintext
const PI : real = 3.14
```
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
• YAKINDU provides a 'Pluggable Type System'
• Type contributions can be provided by plugins
• Type checking is performed on expressions.
• Type inferrer calculates (infers) types of expressions and checks type constraints.
• Type inferrer is extendable in order to implement specific checks.
• Immediate user feedback while editing
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

Parametrization
• 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
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.

<table>
<thead>
<tr>
<th>TrafficLight</th>
</tr>
</thead>
<tbody>
<tr>
<td>- counter: Integer = 0</td>
</tr>
<tr>
<td>- green: Boolean = false</td>
</tr>
<tr>
<td>- red: Boolean = false</td>
</tr>
<tr>
<td>- yellow: Boolean = false</td>
</tr>
</tbody>
</table>

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
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)

```python
1  def simulate(sc: Statechart) {

18  }
```
YAKINDU strategy: first enabled transition is selected. If found no further transitions are tested.

**Enabled:**

```java
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
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)
        }
    }
}
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


• 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 someTestclass for statechart Light_Switch {

}
• Has a unique name

• A testsuite contains at least one reference to a testclass
```

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
// 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)
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
• 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*:
• 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

```java
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)
Hierarchy: Modified Example

Semantics/FLATTEN semantics? (unwanted) non-determinism! determinism!

choose outer transition

choose inner transition

Statemate, Yakindu, ...

Rhapsody, ...
Hierarchical states are an ideal mechanism for hiding complexity.

- Parent states can implement common behavior for its substates.
- Hierarchical event processing reduces the number of transitions.
- Refactoring support: group state into composite.
• 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
• 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:
Exercise 5

Model an interruptible traffic light
Exercise 5 - Requirements

• R7a: police can interrupt autonomous operation.
• R7b: Autonomous operation can be interrupted during any phase indicated by 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 indicated by 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

<table>
<thead>
<tr>
<th>R6: police can interrupt autonomous operation.</th>
<th>An new incoming event police_interrupt triggers a transition to a new state interrupted.</th>
</tr>
</thead>
<tbody>
<tr>
<td>R6a: Autonomous operation can be interrupted during any phase indicated by constant red, yellow and green lights.</td>
<td>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.</td>
</tr>
<tr>
<td>R7: In interrupted mode the yellow light blinks with a constant frequency of 1 Hz. (on -&gt; 0.5s, off 0.5s).</td>
<td>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.</td>
</tr>
<tr>
<td>R8: Police can resume to regular autonomous operation.</td>
<td>A transition triggered by police_interrupt leads from state interrupted to state normal.</td>
</tr>
<tr>
<td>R8a: When regular operation is resumed the traffic light restarts with red (R) light on.</td>
<td>When activating state normal its substate Red is activated by default.</td>
</tr>
</tbody>
</table>
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:
• 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 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 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 complete 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

```
+-----------------+        +------------------+
| normal          |        | interrupted       |
+-----------------+        +------------------+
| Red             |        | Yellow            |
|                 |        |                   |
| after(60) / ^displayGreen | police_interrupt | after(1) / ^displayYellow |
|                 |        |                   |
| Green           |        |                   |
|                 |        |                   |
| after(5) / ^displayRed | police_interrupt | after(1) / ^displayNone |
|                 |        |                   |
| Yellow          |        | Black             |
|                 |        |                   |
| after(55) / ^displayYellow | police_interrupt | after(1) / ^displayYellow |
|                 |        |                   |
+-----------------+        +------------------+
```

- Normal state:
  - Red
    - After 60 seconds, display Green
    - After 5 seconds, display Red
  - Green
    - After 55 seconds, display Yellow
  - Yellow
- Interrupted state:
  - After 1 second, display None
  - Black
- On state:
  - Toggle to display None
  - Off
    - Toggle to display Red
Orthogonality
Orthogonality

Semantics/meaning?

Effective target states:

Recursive!
Parallel (In)Dependence
Parallel (In)Dependence
Parallel (In)Dependence

MyClass
  counter : int

<<behaviour>>

main

/ counter = 0

A
  inp / counter += 2
  inp / counter *= 3

X

B

Y

(A, X)
  counter = 0

(B, X)
  :MyClass
  counter| 2

(B, Y)
  :MyClass
  counter| 6

(A, Y)
  :MyClass
  counter| 0

inp / counter += 2

inp / counter *= 3

inp / counter *= 3

inp / counter += 2

≠
Orthogonality: Communication

- Components can communicate:
  - raising local events:
    ^'<<event name>>
  - INSTATE macro
    INSTATE(<<state location>>)

Input Segment: nmnn
Simulation Algorithm

```plaintext
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

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

<table>
<thead>
<tr>
<th>requirement</th>
<th>modelling approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>R10: a timer displays the remaining time while the light is red or green</td>
<td>The timer is defined in a second region within state on.</td>
</tr>
<tr>
<td>R10a: This timer decreases and displays its value every second.</td>
<td>An internal variable for the counter is introduced. When switching a traffic light phase the counter value is set to the time period of the phase. Additionally, the local events resetTimer, enableTimer, and disableTimer are used to synchronize traffic light phase switches with the timer.</td>
</tr>
<tr>
<td>R10b: The colour of the timer reflects the colour of the traffic light.</td>
<td>When the timer is enabled it checks the active traffic light phase state using active() function.</td>
</tr>
</tbody>
</table>
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
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

```java
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.

```c
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
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

<table>
<thead>
<tr>
<th></th>
<th>Fast</th>
<th>Memory efficient</th>
<th>easy to debug</th>
<th>Easy to understand</th>
</tr>
</thead>
<tbody>
<tr>
<td>Switch / Case</td>
<td>+</td>
<td>+</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>State Transition Table</td>
<td>+</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>State Pattern</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>

⚠️ very simplified illustration
• 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

Files

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

Sample

```
private void exitSequence_main_main_trafficlight_interrupted_blinking_Yellow() {
  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 blinking */

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 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**

```java
interface TrafficLightCtrl {
    in event police_interrupt
    in event toggle
}
```

**Generator**

```java
GeneratorModel for yakindu::java {
    statechart TrafficLightCtrl {
        feature Outlet {
            targetProject = "traffic_light_history"
            targetFolder = "src-gen"
        }
        feature Naming {
            basePackage = "traffic.light"
            implementationSuffix = ""
        }
        feature GeneralFeatures {
            InterfaceService = true
            TimerService = true
            InterfaceObserverSupport = true
        }
        feature SynchronizedWrapper {
            namePrefix = "Synchronized"
            nameSuffix = ""
        }
    }
}
```

**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 onUpdateTimeValueRaised(long value) {
            crossing.getCounterVis().setCounterValue(value);
            repaint();
        }

        @Override
        public void onUpdateTimeColourRaised(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.initState();
}
```

**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**

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

void loop() {
    unsigned long read_pushbut
    if (cycle_count < CYCLE_PERIOD)
        TrafficLightCtrl {
            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
                }
            }
        }
    }
}
```

**Generator**

```c
GeneratorModel for yakindu::c {

    if (cycle_count < CYCLE_PERIOD) {
        last_cycle_time = cycle_count;
    }
}
```

**Button Cc**

```c
void read_pushbut()
    int pin_value
    if (pin_value>
        button->la;
    }
    if ((millis()<
        if (pin_value>
            button->;
        button->;
    }
    button->debounce_state = pin_value;
```
Semantic Choices
 enabled events: [\textit{inc\_one}, \textit{inc\_two}]
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.

---

Semantic Options

Revisiting the Example

enabled events: \([\text{inc\_one}, \text{inc\_two}]\)

concurrency: single

event lifeline: next combo step

assignment: RHS small step

\(- \rightarrow \langle \{t1\}, \{t3\}, \{t5\} \rangle \) and

\( \langle \{t3\}, \{t1\}, \{t5\} \rangle \)

event lifeline: present in remainder

\(- \rightarrow \langle \{t1\}, \{t5\}, \{t3\} \rangle \) becomes possible

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

<table>
<thead>
<tr>
<th></th>
<th>Rhapsody</th>
<th>Statemate</th>
<th>(Default) SCCD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Big Step Maximality</td>
<td>Take Many</td>
<td>Take Many</td>
<td>Take Many</td>
</tr>
<tr>
<td>Internal Event Lifeline</td>
<td>Queue</td>
<td>Next Combo Step</td>
<td>Queue</td>
</tr>
<tr>
<td>Input Event Lifeline</td>
<td>First Combo Step</td>
<td>First Combo Step</td>
<td>First Combo Step</td>
</tr>
<tr>
<td>Priority</td>
<td>Source-Child</td>
<td>Source-Parent</td>
<td>Source-Parent</td>
</tr>
<tr>
<td>Concurrency</td>
<td>Single</td>
<td>Single</td>
<td>Single</td>
</tr>
</tbody>
</table>

Child-first vs Parent-first
Event-driven vs Cycle-based
In which order are transitions evaluated in a HSM?

**Parent-first** execution tries to take transitions top down

**Child-first** execution tries to take transition bottom up

Child transitions **overwrite** parent behavior

Parent transitions **shadow** child behavior
The behavior of state machines are executed in single ‘run-to-completion’ steps.

Event-driven execution runs a single RTC step for each incoming event.

Cycle-based execution runs RTC steps isochronously.

Execution depends on events.

Execution is independent of events: \( X=10 \).

One event visible in RTC step.

0..n events visible in RTC step.
• 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

```
internal events

A -> B: e / raise i; raise j
  B -> C

raise e

A -> B: e / raise i; raise j
  B -> C

process i

A -> B: e / raise i; raise j
  B -> C

process j

A -> B: e / raise i; raise j
  B -> C
```

Event-driven: event queuing
Composition
Composition of Statecharts

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

crossing control

Off

toggleOnOff

toggleOnOff

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

inner region

prepare release A

entry / trafficLightA.raiseLock; trafficLightB.raiseLock

after 10s

every 100ms
[trafficLightB.isRaisedLocked]

release B

entry / trafficLightB.raiseRelease

after 10s

every 100ms
[trafficLightA.isRaisedLocked]

prepare release B

entry / trafficLightA.raiseLock; trafficLightB.raiseLock

entry / trafficLightA.raiseRelease

release A

entry / trafficLightA.raiseRelease

normal

ON/OFF

POLICE INTERRUPT

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...

Communication: Event Scopes
SCCD Compiler

SCCD Documentation

SCCD is a language that combines the Statecharts language with Class Diagrams. It allows users to model complex, timed, autonomous, reactive, and dynamic 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 modeled 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

- Installation
  - Download
  - Dependencies
  - SCCD Installation
- Language Features
  - Top-Level Elements
  - Class Diagram Concepts
  - Statechart Concepts
  - Executable Content
  - Macros
  - Object Manager
- Compiler
- Runtime Platforms
  - Threads
  - Eventloop
  - Gameloop
- Examples
  - Timer
  - Traffic Lights
- Semantic Options
  - Big Step Maximality
  - Internal Event Lifeline
  - Input Event Lifeline
  - Priority
  - Concurrency
- Socket Communication
  - Initialization
  - Input Events
  - Output Events
  - HTTP client/server
- Internal Documentation
  - Statecharts Core

## References

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