An Introduction to Statecharts Modelling and Simulation

Simon Van Mierlo

simon.vanmierlo@uantwerpen.be

Hans Vangheluwe

hans.vangheluwe@uantwerpen.be

Universiteit Antwerpen

FLANDERS MAKE
MANUFACTURING INNOVATION NETWORK

McGill

INTRODUCTION

STATECHARTS BASICS

YAKINDU IN DEPTH

ADVANCED CONCEPTS

Reactive Systems

- Comp … ctive (to events), timed, concurrent …
- In con … national systems, whic …
  event … 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)?

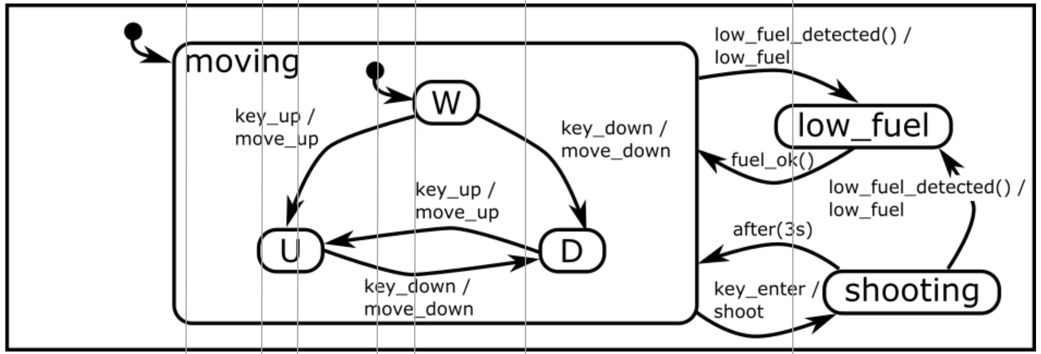
"Nontrivial software written with threads, semaphores, and mutexes are incomprehensible to humans"[1]

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

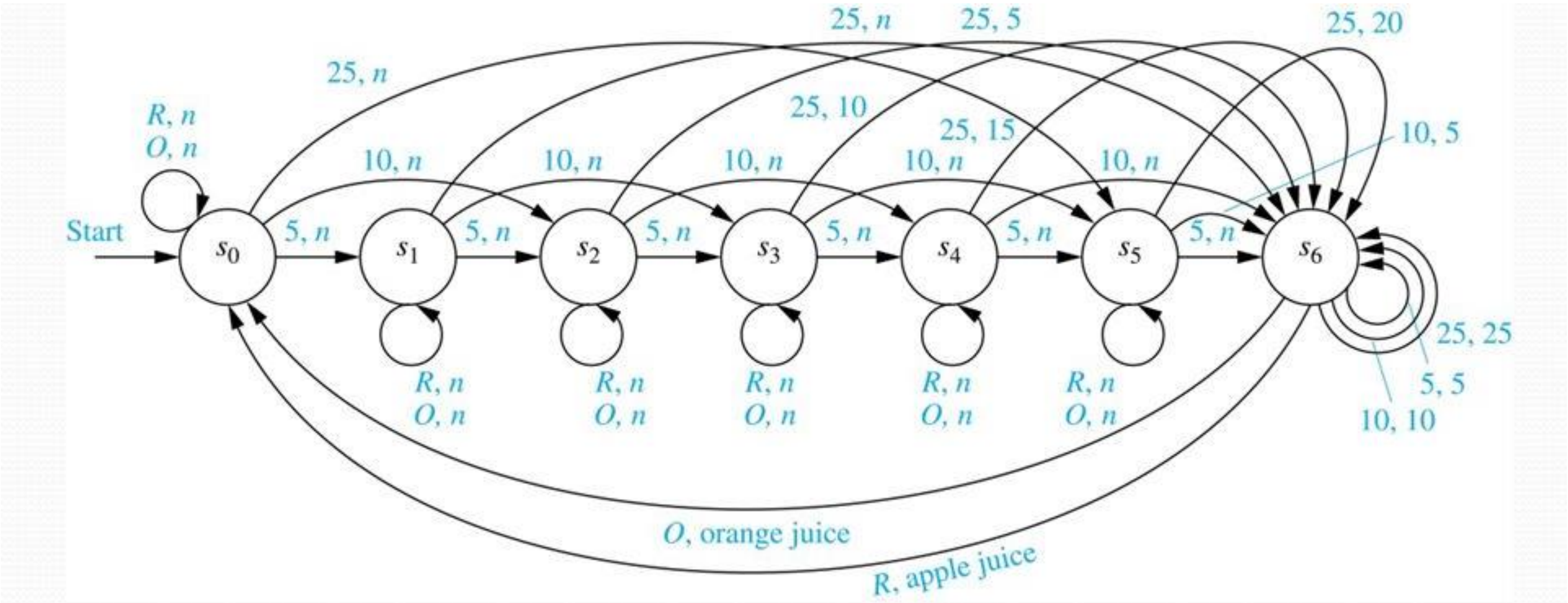
[1]E. A. Lee, "The problem with threads," in *Computer*, vol. 39, no. 5, pp. 33-42, May 2006.

Discrete-Event Abstraction

behavioural
model

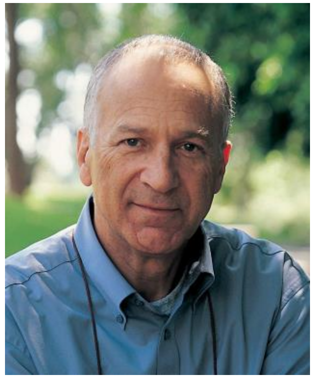


State Diagrams



- Non-modular: hierarchical decompositition (orthogonal/depth) not possible
- State space limited (positive: analysability, negative: expressitivity)
- Becomes too large too quickly to be usable

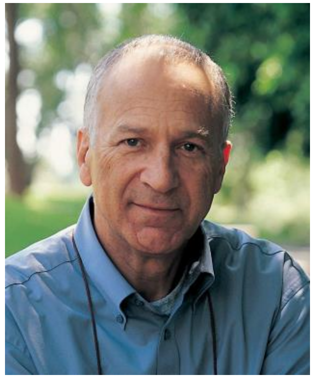
Statecharts History

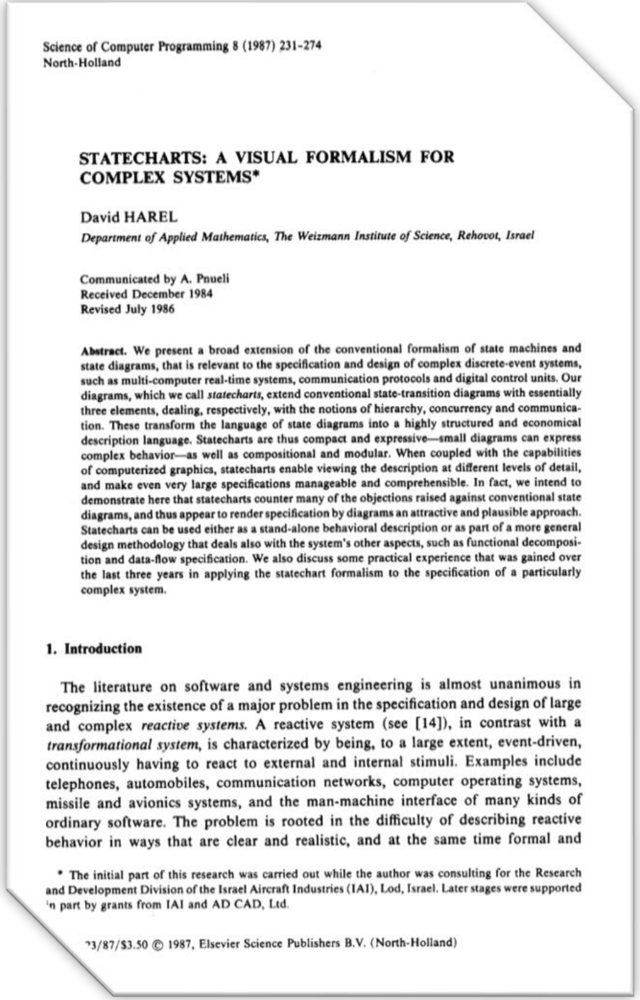


- Introduced by David Harel in 1987[1]
- Notation based on higraphs = hypergraphs + Euler diagrams
- Semantics extend deterministic finite state automata with:
  - Depth (Hierarchy)
  - Orthogonality
  - Broadcast Communication
  - Time
  - History
  - Syntactic sugar, such as enter/exit actions

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

Statecharts History





Science of Computer Programming 8 (1987) 231-274
North-Holland

STATECHARTS: A VISUAL FORMALISM FOR COMPLEX SYSTEMS*

David HAREL
Department of Applied Mathematics, The Weizmann Institute of Science, Rehovot, Israel

Communicated by A. Pnueli
Received December 1984
Revised July 1986

Abstract. We present a broad extension of the conventional formalism of state machines and state diagrams, that is relevant to the specification and design of complex discrete-event systems, such as multi-computer real-time systems, communication protocols and digital control units. Our diagrams, which we call *statecharts*, extend conventional state-transition diagrams with essentially three elements, dealing, respectively, with the notions of hierarchy, concurrency and communication. These transform the language of state diagrams into a highly structured and economical description language. Statecharts are thus compact and expressive—small diagrams can express complex behavior—as well as compositional and modular. When coupled with the capabilities of computerized graphics, statecharts enable viewing the description at different levels of detail, and make even very large specifications manageable and comprehensible. In fact, we intend to demonstrate here that statecharts counter many of the objections raised against conventional state diagrams, and thus appear to render specification by diagrams an attractive and plausible approach. Statecharts can be used either as a stand-alone behavioral description or as part of a more general design methodology that deals also with the system's other aspects, such as functional decomposition and data-flow specification. We also discuss some practical experience that was gained over the last three years in applying the statechart formalism to the specification of a particularly complex system.

1. Introduction

The literature on software and systems engineering is almost unanimous in recognizing the existence of a major problem in the specification and design of large and complex *reactive systems*. A reactive system (see [14]), in contrast with a *transformational system*, is characterized by being, to a large extent, event-driven, continuously having to react to external and internal stimuli. Examples include telephones, automobiles, communication networks, computer operating systems, missile and avionics systems, and the man-machine interface of many kinds of ordinary software. The problem is rooted in the difficulty of describing reactive behavior in ways that are clear and realistic, and at the same time formal and

* The initial part of this research was carried out while the author was consulting for the Research and Development Division of the Israel Aircraft Industries (IAI), Lod, Israel. Later stages were supported 'n part by grants from IAI and AD CAD, Ltd.

ON VISUAL FORMALISMS

The higraph, a general kind of diagramming object, forms a visual formalism of topological nature. Higraphs are suited for a wide array of applications to databases, knowledge representation, and, most notably, the behavioral specification of complex concurrent systems using the higraph-based language of statecharts.

DAVID HAREL

Visualizing information, especially information of complex and intricate nature, has for many years been the subject of considerable work by many people. The information that interests us here is nonquantitative, but rather, of a structural, set-theoretical, and relational nature. This should be contrasted with the kinds of quantitative information discussed at length in [43] and [46]. Consequently, we shall be interested in diagrammatic paradigms that are essentially topological in nature, not geometric, terming them *topovisual* in the sequel.

Two of the best known topo-visual formalisms have their roots in the work of the famous Swiss mathematician Leonhard Euler (1707–1783). The first, of course, is the formalism of graphs, and the second is the notion of *Euler circles*, which later evolved into *Venn diagrams*. Graphs are implicit in Euler's celebrated 1736 paper, in which he solved the problem of the bridges of Königsberg [12]. (An English translation appears in [3].) Euler circles first appear in letters written by Euler in the early 1760s [13], and were modified to improve their ability to represent logical propositions by John Venn in 1880 [48, 49]. (See [19, chap. 2] for more information.[1])

A graph, in its most basic form, is simply a set of points, or nodes, connected by edges or arcs. Its role is to represent a (single) set of elements S and some binary relation R on them. The precise meaning of the relation R is part of the application and has little to do with the mathematical properties of the graph itself. Certain restrictions on the relation R yield special classes of graphs that are of particular interest, such as ones that are connected, directed, acyclic, planar, or bipartite. There is no need to elaborate on the use of graphs in computer science—they are used extensively in virtually all branches of the field. The elements represented by the nodes in these applications range from the most concrete (e.g., physical gates in a circuit diagram) to the most abstract (e.g., complexity classes in a classification schema), and the edges have been used to represent almost any conceivable kind of relation, including ones of temporal, causal, functional, or epistemological nature. Obviously, graphs can be modified to support a number of different kinds of nodes and edges, representing different kinds of elements and relationships.

A somewhat less widely used extension of graphs is the formalism of *hypergraphs* (see, e.g., [1]), though these are also finding applications in computer science, mainly in database theory (see [14], [15], and [31]). A hypergraph is a graph in which the relation being specified is not necessarily binary; in fact, it need not even be of fixed arity. Formally, an edge no longer connects a pair of nodes, but rather a subset thereof. This makes hypergraphs somewhat less amenable to visual representation, but various ways of overcoming this difficulty can be conceived (see Figure 1). In analogy with graphs, several special kinds of hypergraphs are of particular interest, such as directed or acyclic.

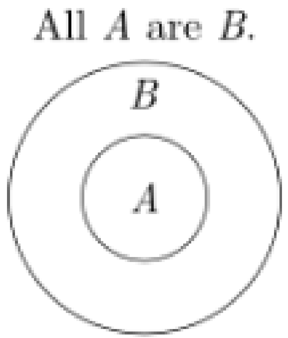
It is important to emphasize that the information

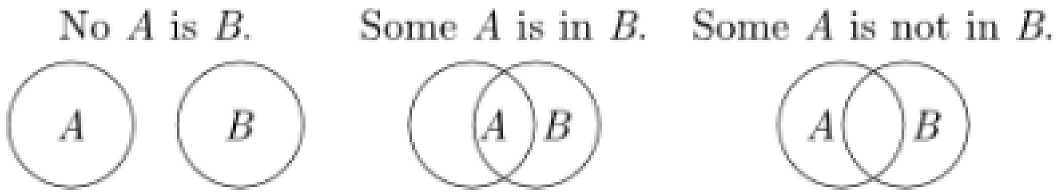
[1] Interestingly, both these topo-visual achievements of Euler were carried out during the period in which he could see with one eye only. (Euler lost sight in his right eye in 1735, and in the left around 1766.) It is tempting to attribute this in part to the fact that the lack of stereoscopic vision reduces one's ability to estimate size and distance, possibly causing a sharper awareness of topological features.

Part of this work was carried out while the author was at the Computer Science Department of Carnegie-Mellon University, Pittsburgh, Pennsylvania.

Higraphs

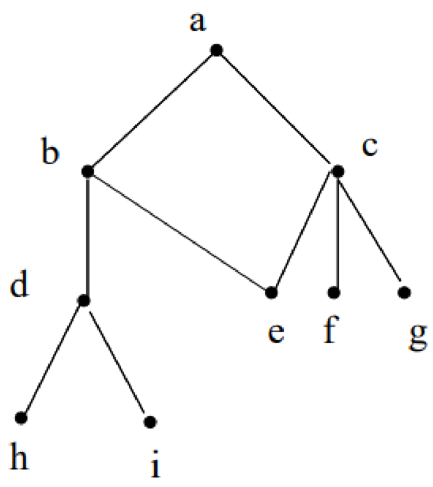
Euler Diagrams



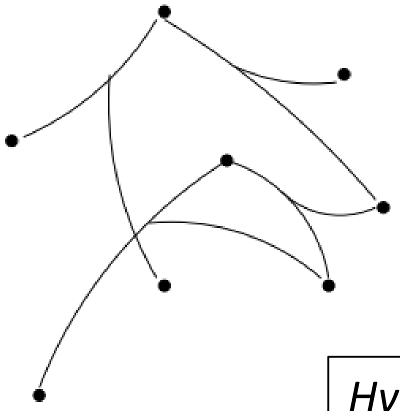


topological notions for set union, difference, intersection

Hypergraphs



a graph



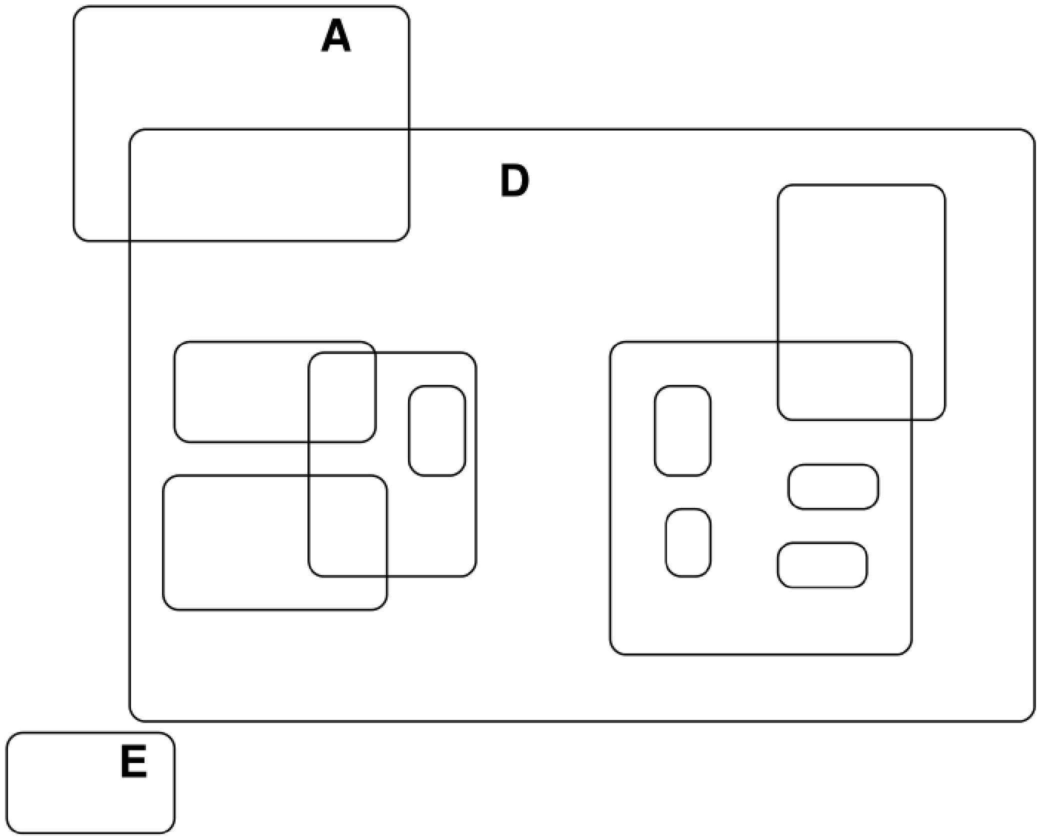
a hypergraph

topological notion (syntax): connectedness

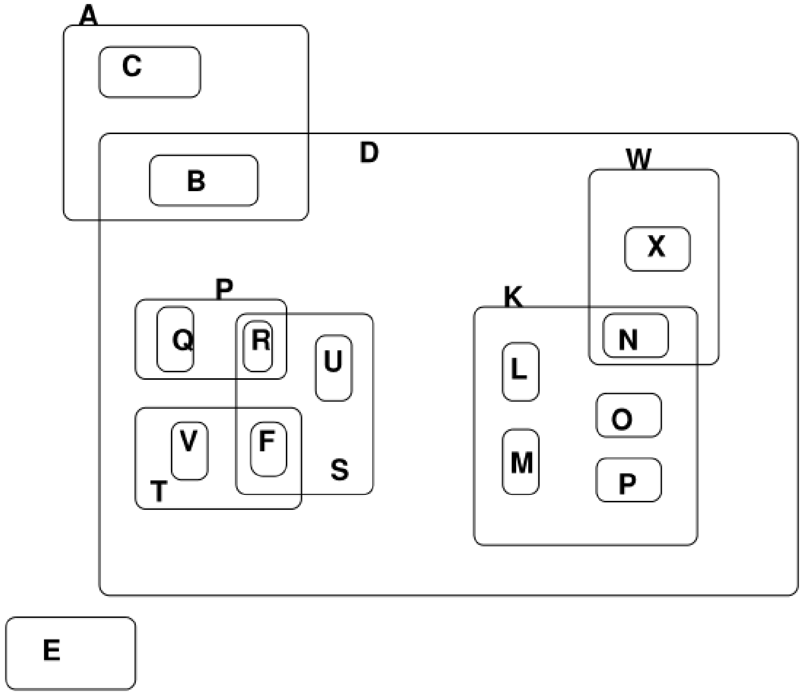
Hyperedges: ⊆ 2X (undirected), ⊆ 2X ×2X (directed).

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

Blobs: set *inclusion*, not *membership*

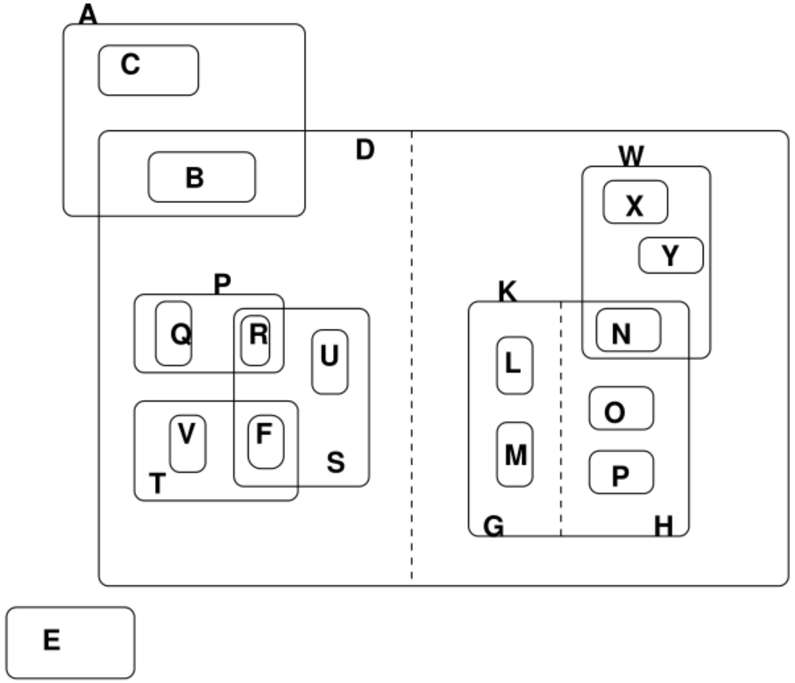


Unique Blobs (atomic *sets*, no *intersection*)



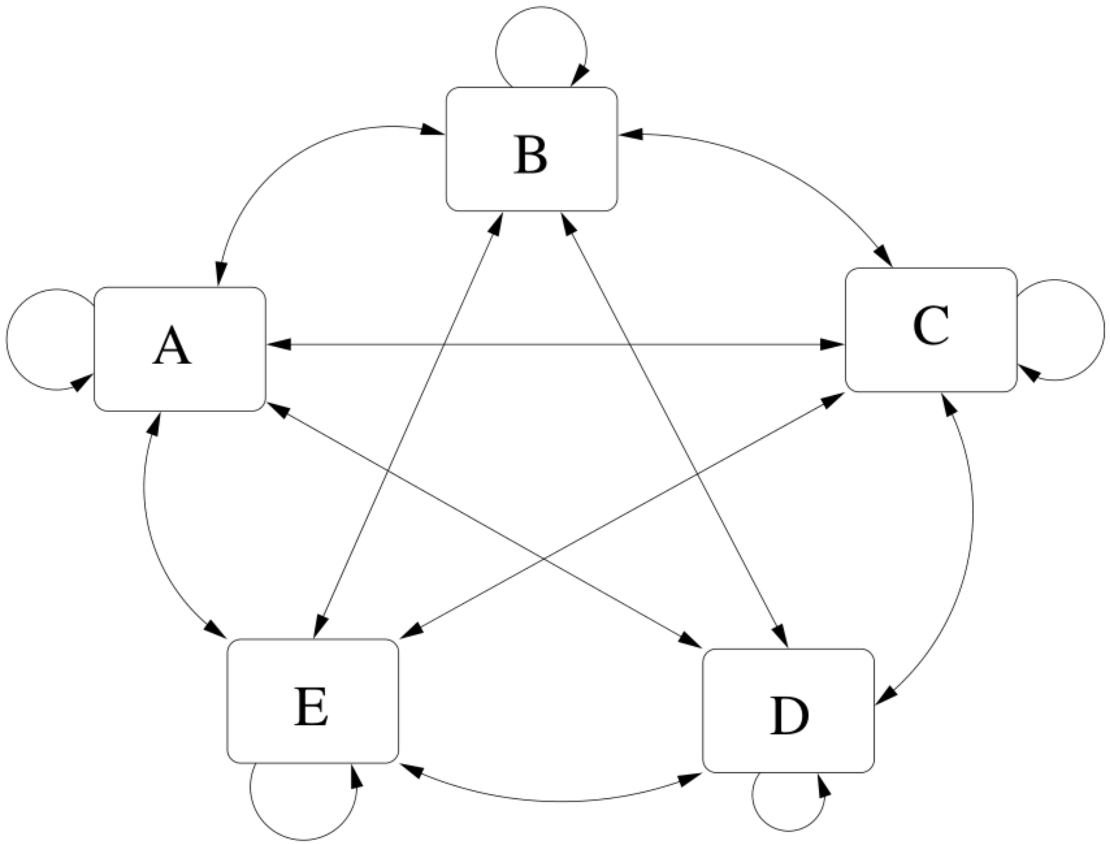
- Atomic blobs are identifiable sets
- Other blobs are union of enlclosed set (*e.g.*, K = L *U* M *U* N *U* O *U* P)

Unordered Cartesian Product: *Orthogonal* Components

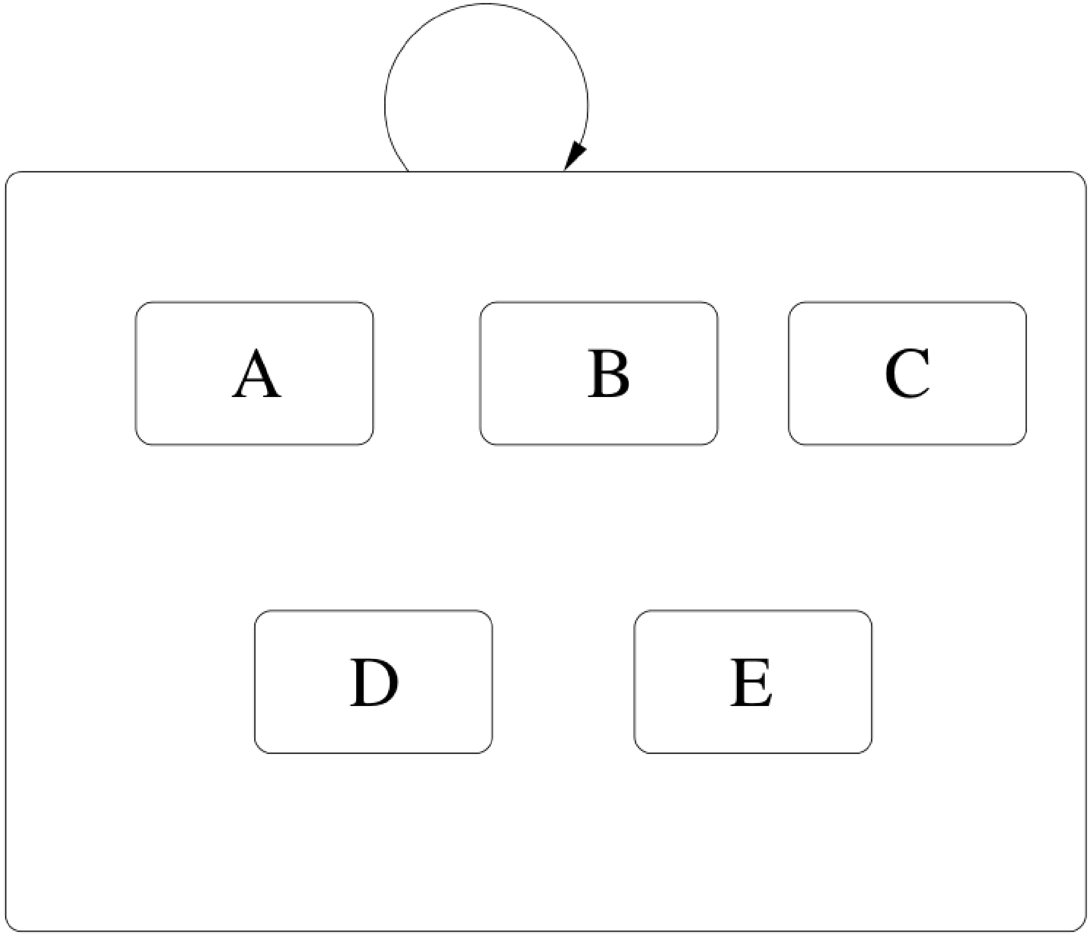


- K = G *x* H = (L *U* M) *x* (N *U* O *U* P)

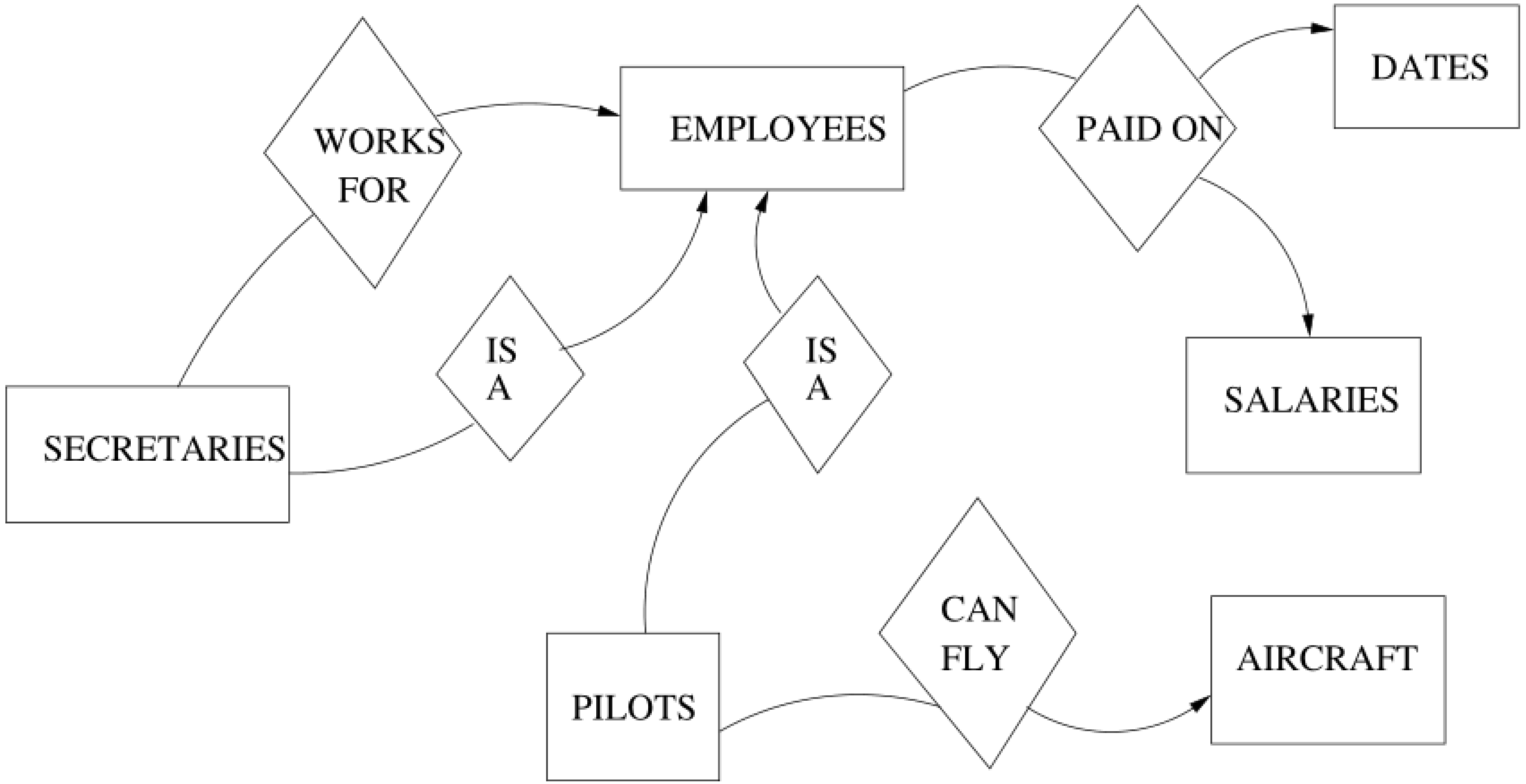
Clique Example



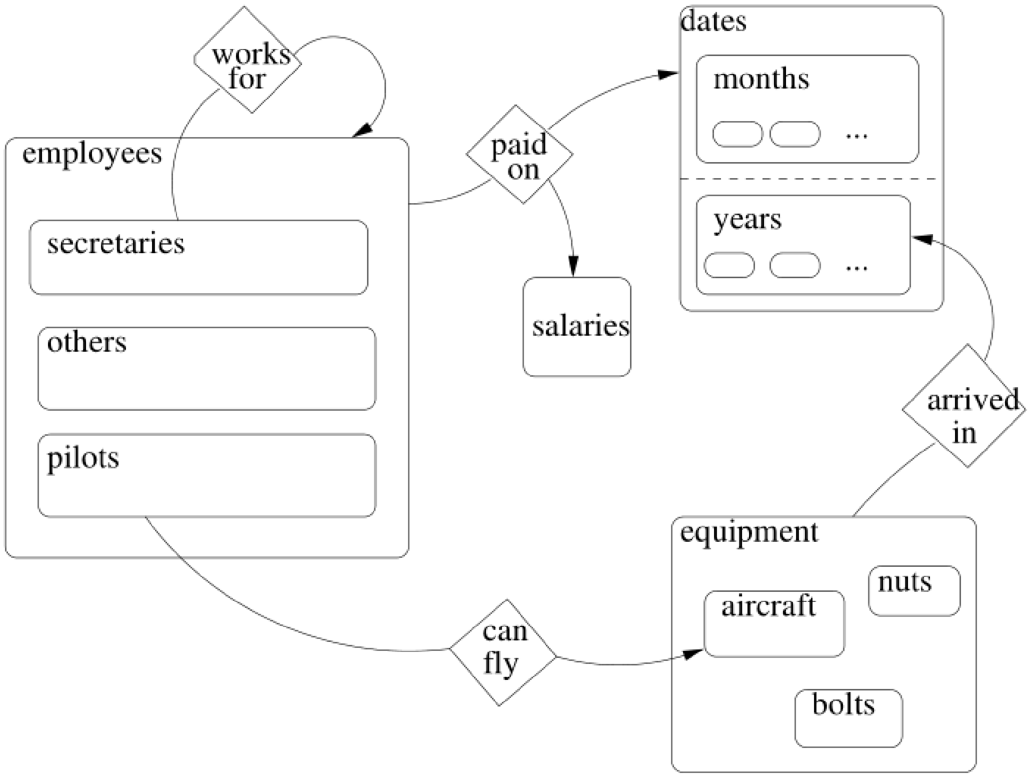
Clique: fully connected semantics



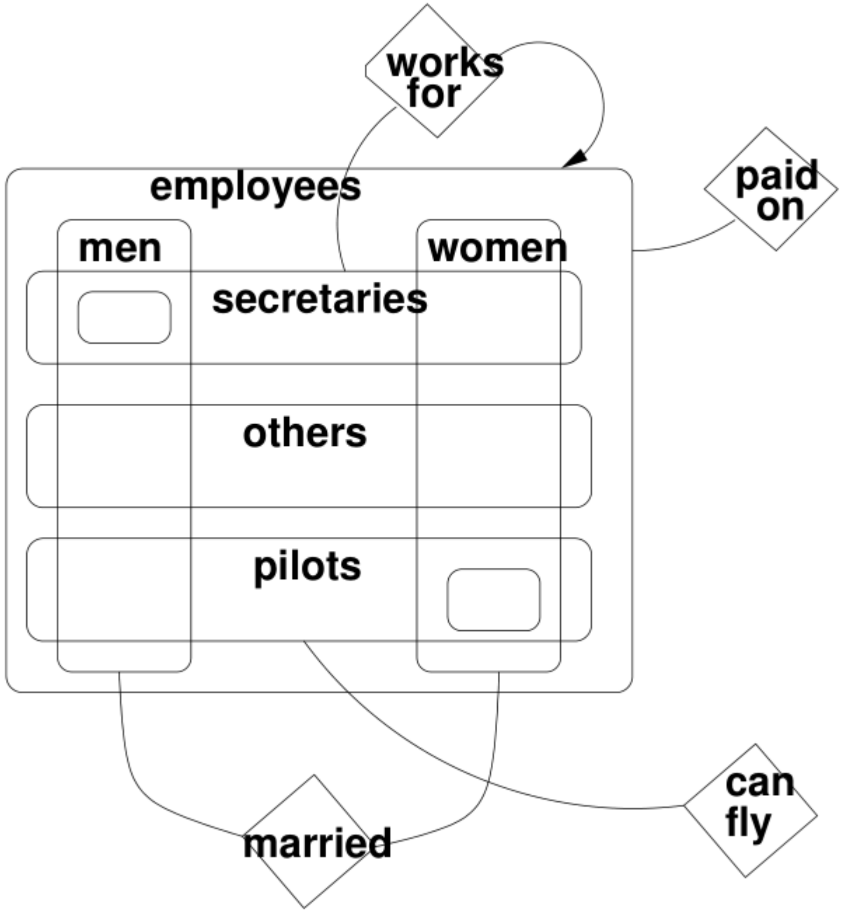
Entity Relationship Diagram



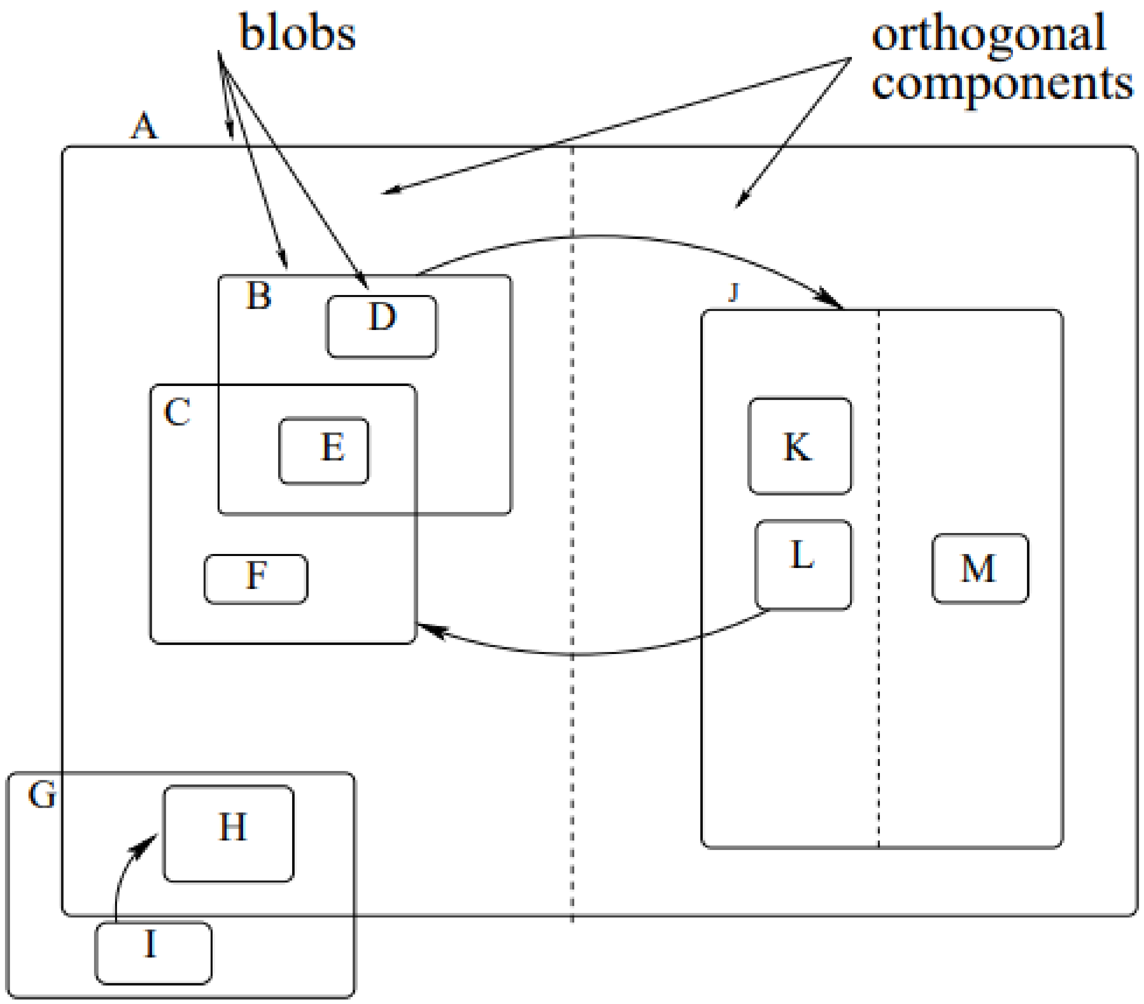
Higraph version of E-R diagram



Extending E-R Diagram



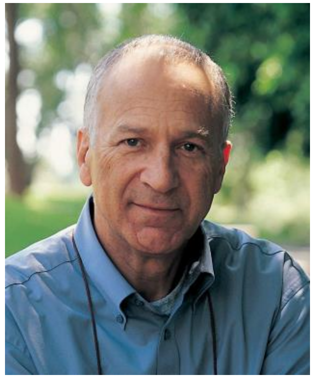
Simple Higraph



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, …

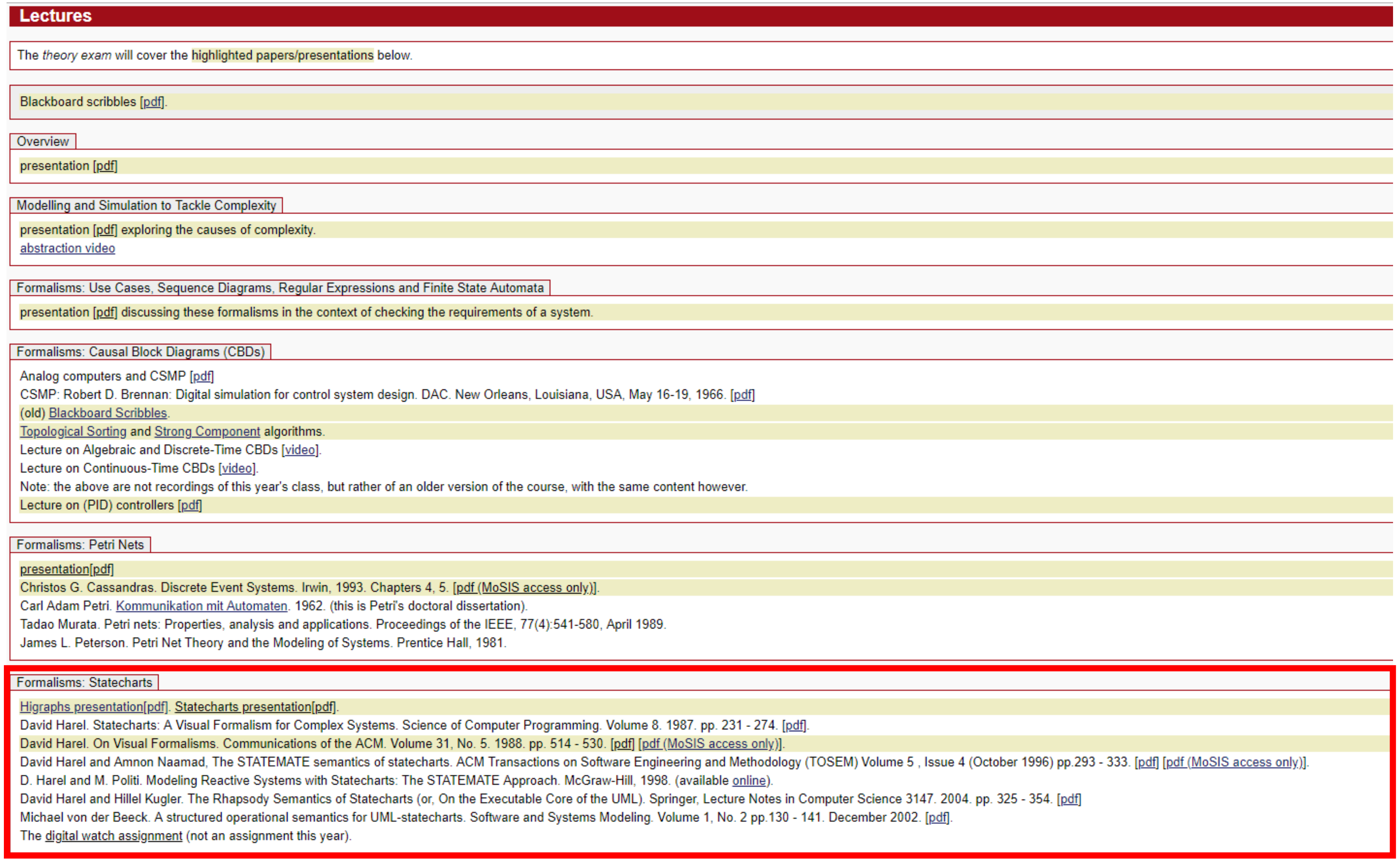
Statecharts History



- Introduced by David Harel in 1987[1]
- Notation based on higraphs = hypergraphs + Euler diagrams
- Semantics extend deterministic finite state automata with:
  - Depth (Hierarchy)
  - Orthogonality
  - Broadcast Communication
  - Time
  - History
  - Syntactic sugar, such as enter/exit actions

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

Statecharts History: Website



Lectures

The *theory exam* will cover the highlighted papers/presentations below.

Blackboard scribbles [pdf].

Overview

presentation [pdf]

Modelling and Simulation to Tackle Complexity

presentation [pdf] exploring the causes of complexity.
abstraction video

Formalisms: Use Cases, Sequence Diagrams, Regular Expressions and Finite State Automata

presentation [pdf] discussing these formalisms in the context of checking the requirements of a system.

Formalisms: Causal Block Diagrams (CBDs)

Analog computers and CSMP [pdf]
CSMP: Robert D. Brennan: Digital simulation for control system design. DAC. New Orleans, Louisiana, USA, May 16-19, 1966. [pdf]
(old) Blackboard Scribbles.
Topological Sorting and Strong Component algorithms.
Lecture on Algebraic and Discrete-Time CBDs [video].
Lecture on Continuous-Time CBDs [video].
Note: the above are not recordings of this year's class, but rather of an older version of the course, with the same content however.
Lecture on (PID) controllers [pdf]

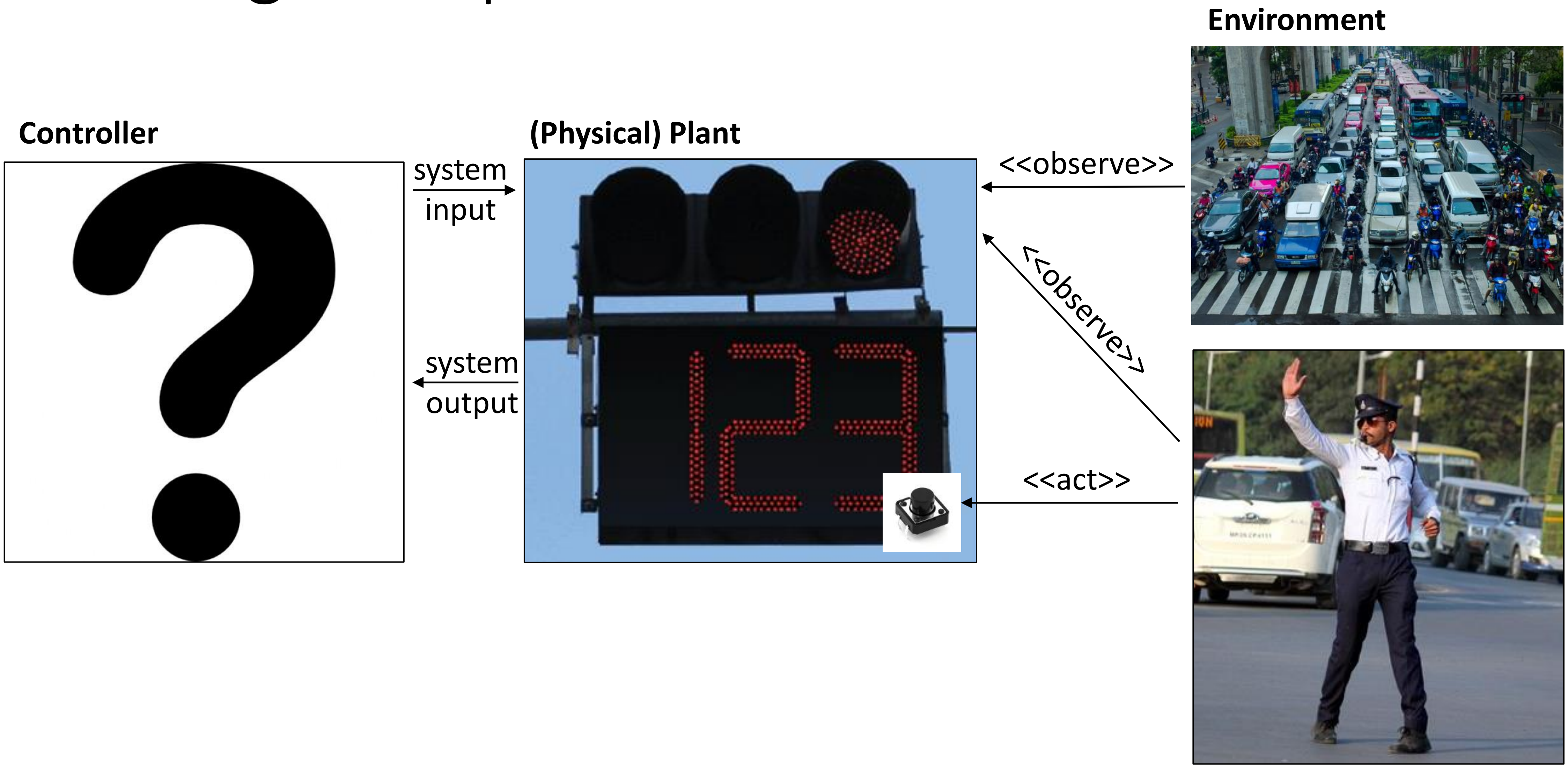
Formalisms: Petri Nets

presentation[pdf]
Christos G. Cassandras. Discrete Event Systems. Irwin, 1993. Chapters 4, 5. [pdf (MoSIS access only)].
Carl Adam Petri. Kommunikation mit Automaten. 1962. (this is Petri's doctoral dissertation).
Tadao Murata. Petri nets: Properties, analysis and applications. Proceedings of the IEEE, 77(4):541-580, April 1989.
James L. Peterson. Petri Net Theory and the Modeling of Systems. Prentice Hall, 1981.

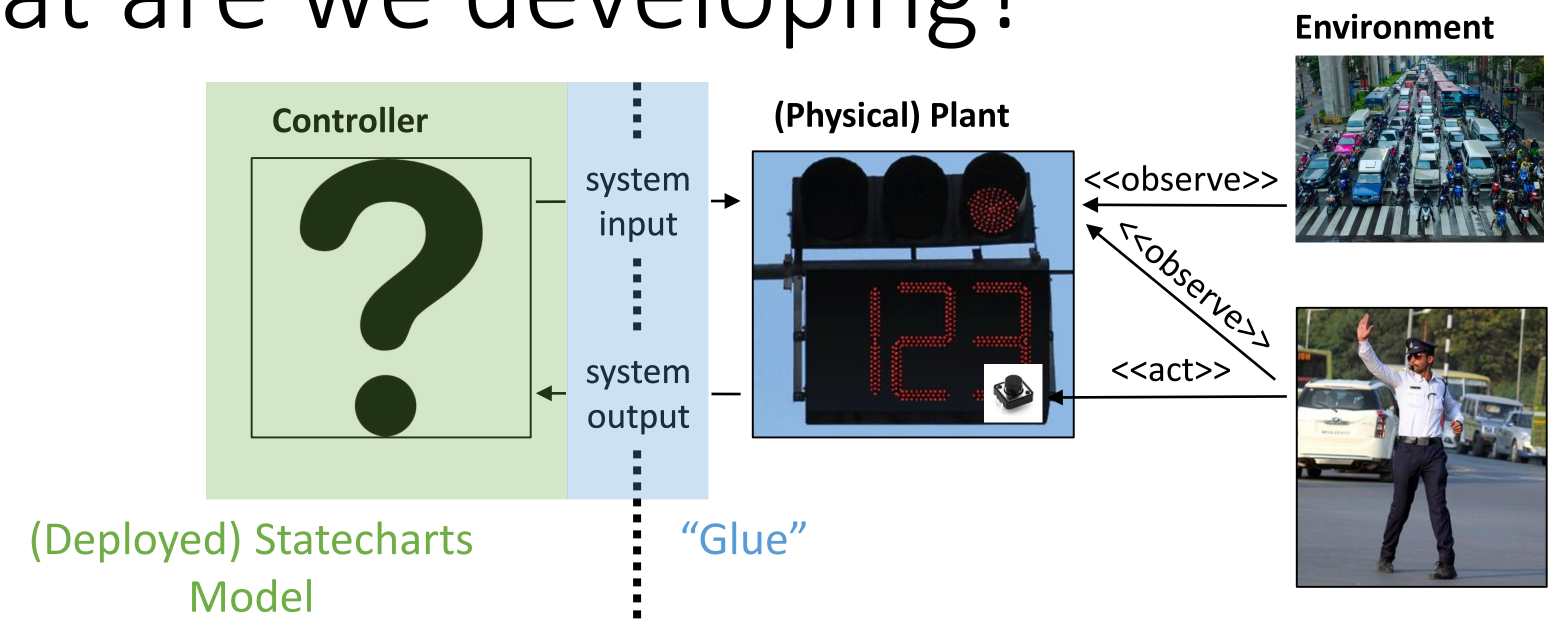
Formalisms: Statecharts

Higraphs presentation[pdf]. Statecharts presentation[pdf].
David Harel. Statecharts: A Visual Formalism for Complex Systems. Science of Computer Programming. Volume 8. 1987. pp. 231 - 274. [pdf].
David Harel. On Visual Formalisms. Communications of the ACM. Volume 31, No. 5. 1988. pp. 514 - 530. [pdf] [pdf (MoSIS access only)].
David Harel and Amnon Naamad, The STATEMATE semantics of statecharts. ACM Transactions on Software Engineering and Methodology (TOSEM) Volume 5 , Issue 4 (October 1996) pp.293 - 333. [pdf] [pdf (MoSIS access only)].
D. Harel and M. Politi. Modeling Reactive Systems with Statecharts: The STATEMATE Approach. McGraw-Hill, 1998. (available online).
David Harel and Hillel Kugler. The Rhapsody Semantics of Statecharts (or, On the Executable Core of the UML). Springer, Lecture Notes in Computer Science 3147. 2004. pp. 325 - 354. [pdf]
Michael von der Beeck. A structured operational semantics for UML-statecharts. Software and Systems Modeling. Volume 1, No. 2 pp.130 - 141. December 2002. [pdf].
The digital watch assignment (not an assignment this year).

Running Example



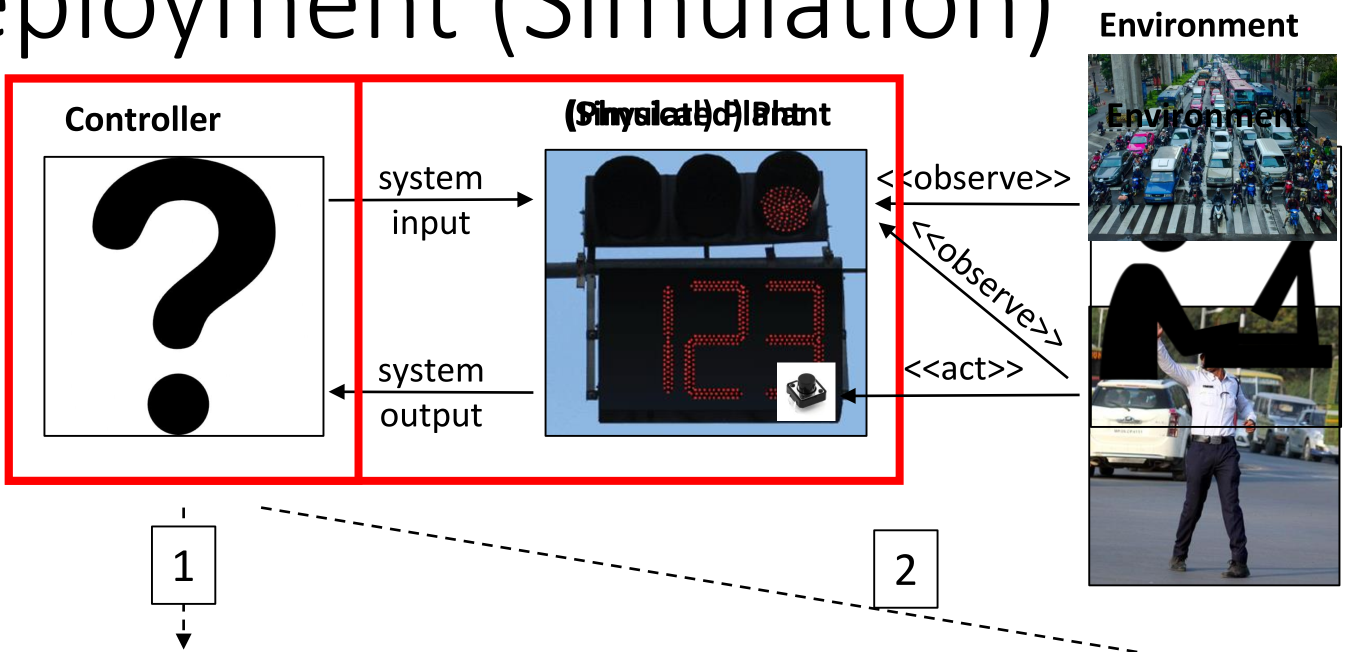
What are we developing?

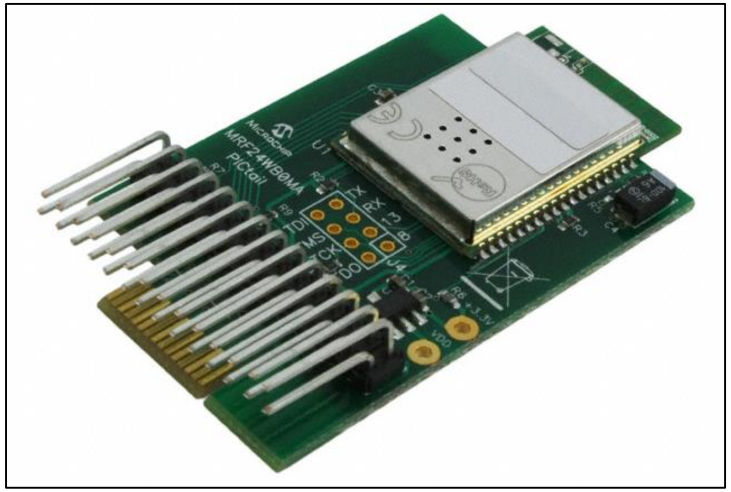


- Autonomous (timed) behaviour
- Interrupt logic
- Orthogonal (traffic light/timer) behaviour

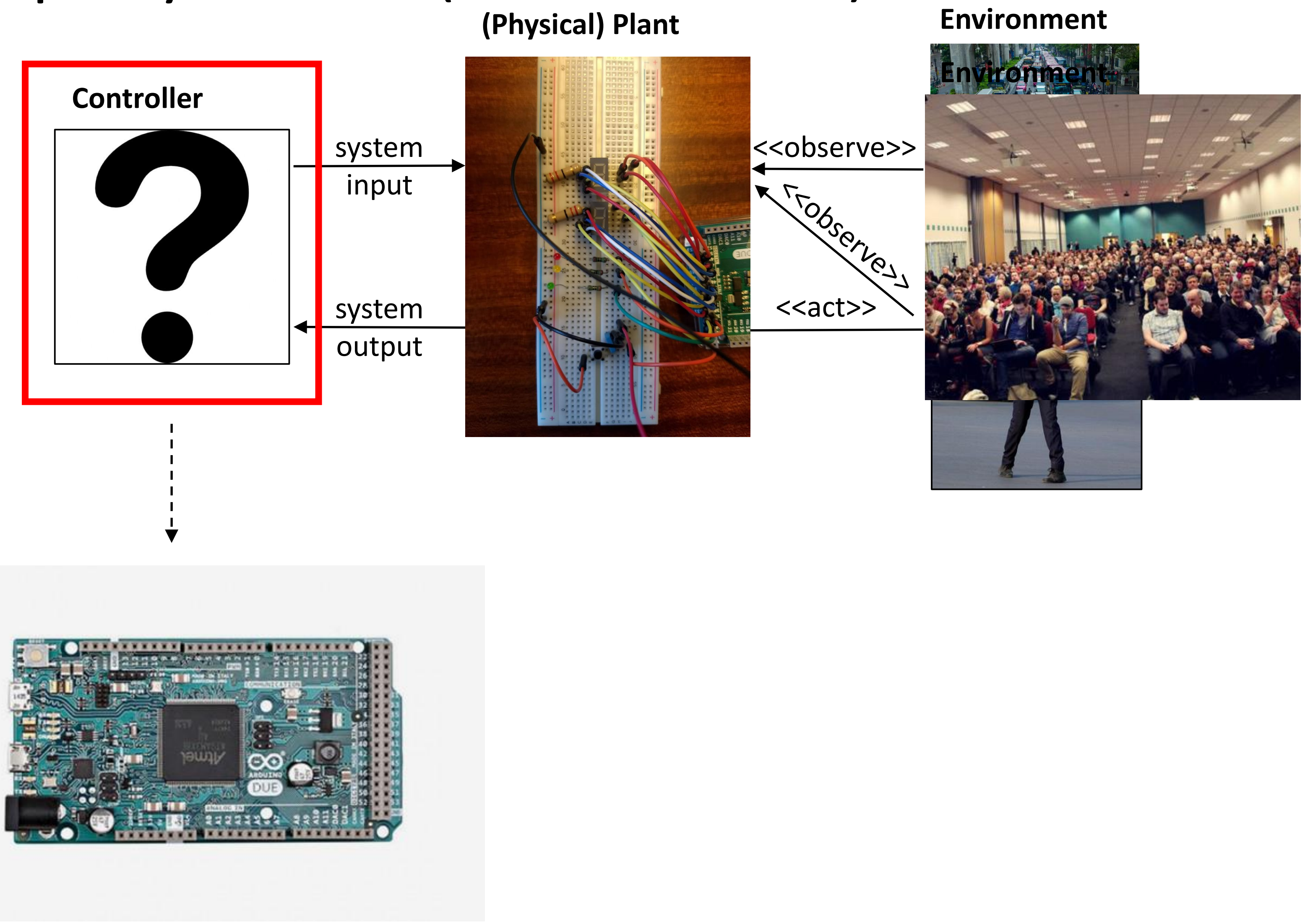
- Turn on/off traffic lights (red/green/yellow)
- Display counter value (three-digit)
- Change counter colour (red/green)
- Sense button presses

Deployment (Simulation)

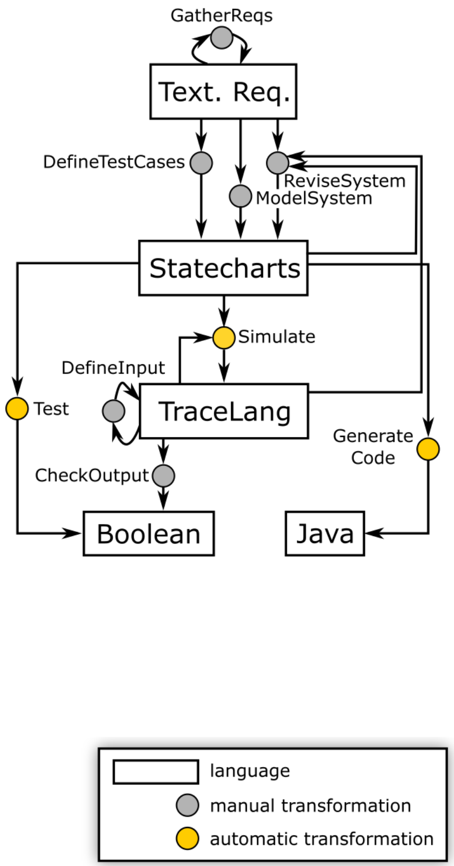


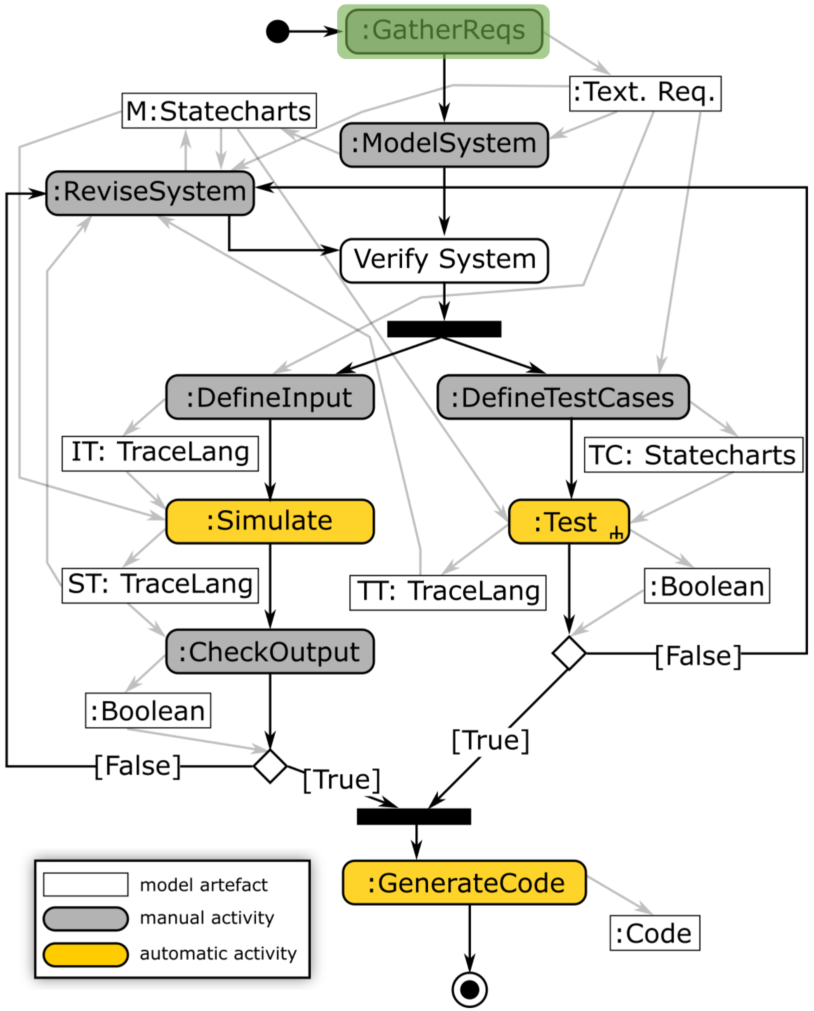


Deployment (Hardware)



Workflow

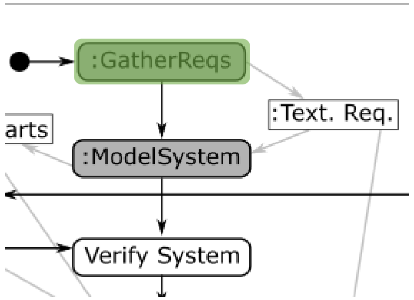




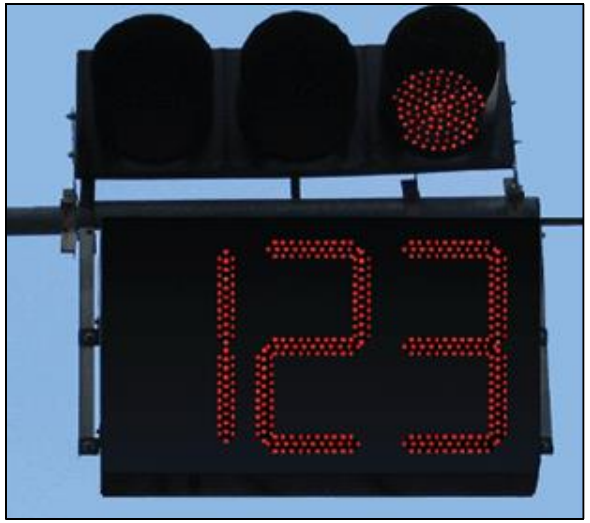
[3] Hans Vangheluwe and Ghislain C. Vansteenkiste. A multi-paradigm modeling and simulation methodology: Formalisms and languages. In European Simulation Symposium (ESS), pages 168-172. Society for Computer Simulation International (SCS), October 1996. Genoa, Italy.

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

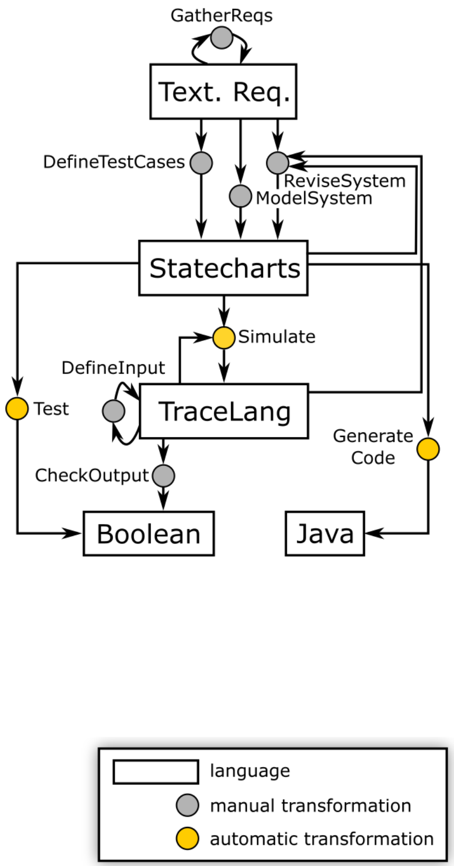
Requirements

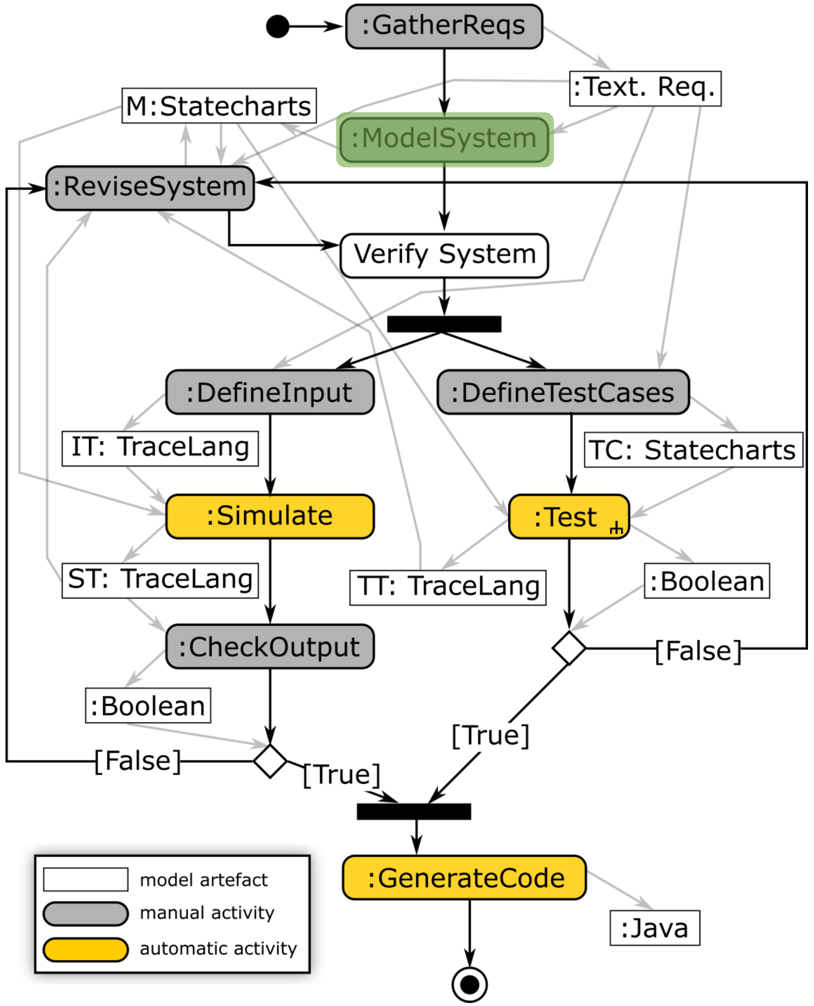


- R1: three differently coloured lights: red, green, yellow
- R2: at most one light is on at any point in time
- R3: at system start-up, the red light is on
- R4: cycles through red on, green on, and yellow on
- R5: red is on for 60s, green is on for 55s, yellow is on for 5s
- R6: police can interrupt autonomous operation
  - Result = blinking yellow light (on -> 1s, off -> 1s)
- R7: police can resume an interrupted traffic light
  - Result = light which was on at time of interrupt is turned on again
- R8: 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.



Workflow


GatherReqs
Text. Req.
DefineTestCases
ReviseSystem
ModelSystem
Statecharts
Simulate
DefineInput
Test
TraceLang
Generate Code
CheckOutput
Boolean
Java
language
manual transformation
automatic transformation

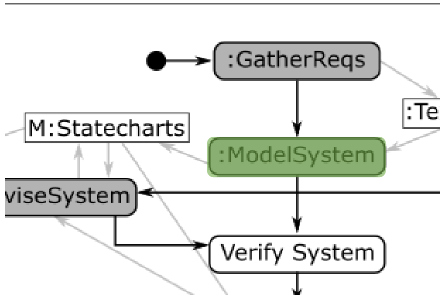

:GatherReqs
:Text. Req.
M:Statecharts
:ModelSystem
:ReviseSystem
Verify System
:DefineInput
:DefineTestCases
IT: TraceLang
TC: Statecharts
:Simulate
:Test
ST: TraceLang
TT: TraceLang
:Boolean
:CheckOutput
[False]
:Boolean
[True]
[False]
[True]
:GenerateCode
:Java
model artefact
manual activity
automatic activity

INTRODUCTION

STATECHARTS BASICS

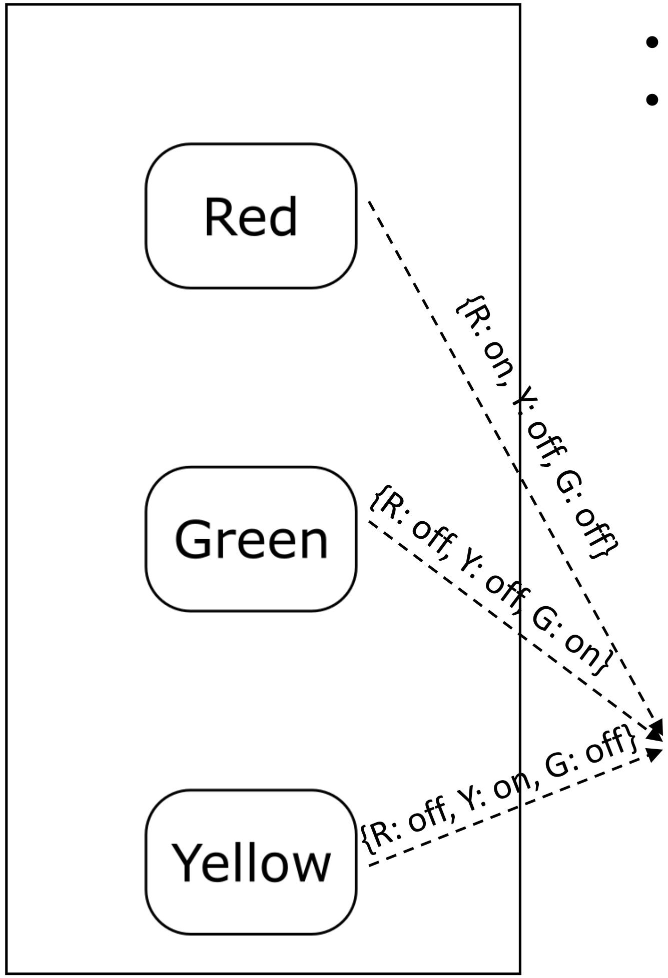
YAKINDU IN DEPTH

ADVANCED CONCEPTS

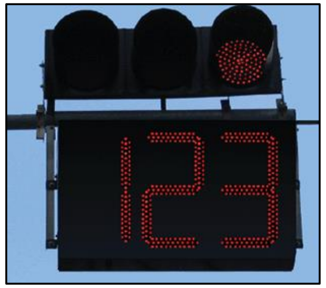


States

- R1: three differently coloured lights: red (R), green (G), yellow (Y)
- R2: at most one light is on at any point in time



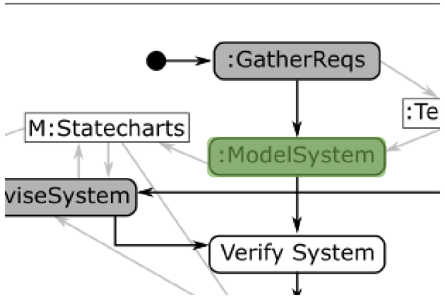
(Simulated) Plant

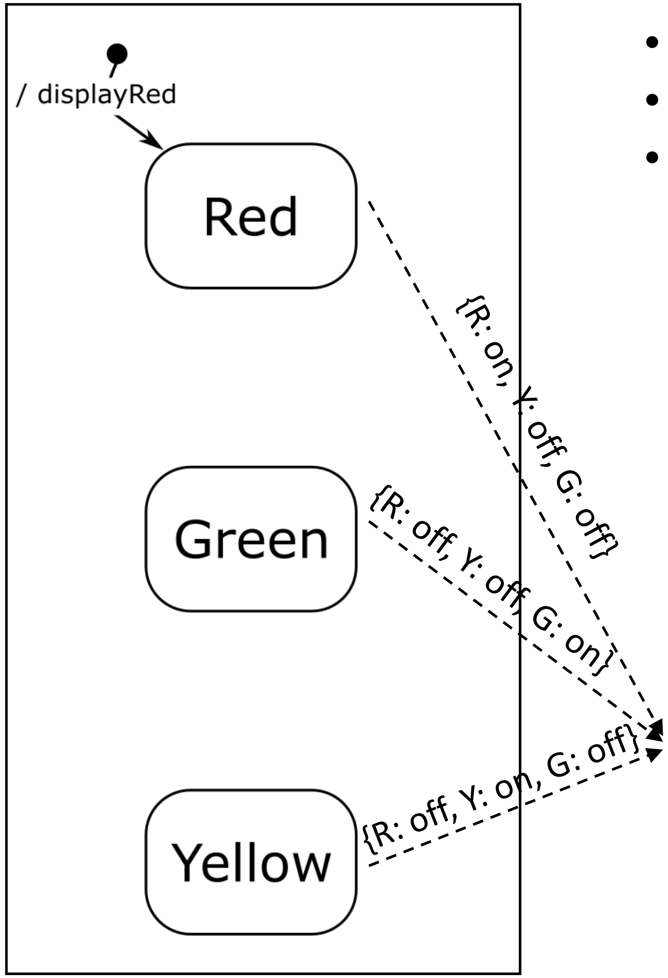


<<observe>>

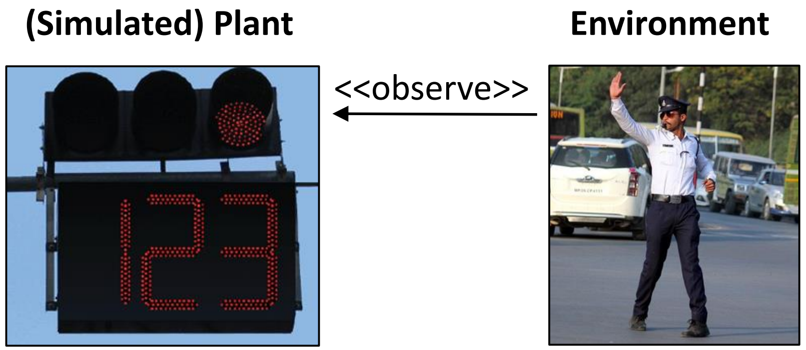
Environment

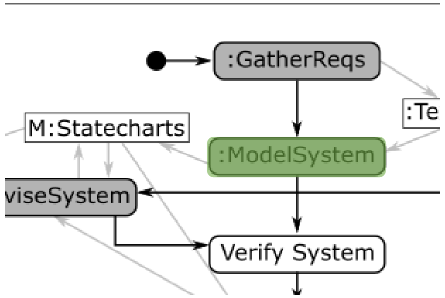
Default State



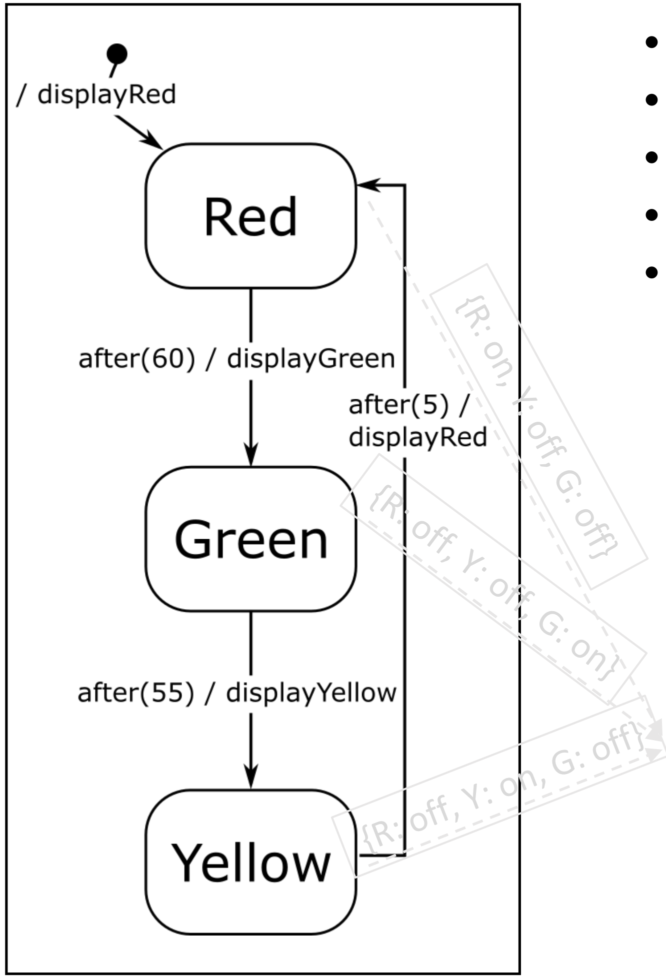


- R1: three differently coloured lights: red (R), green (G), yellow (Y)
- R2: at most one light is on at any point in time
- R3: at system start-up, the red light is on





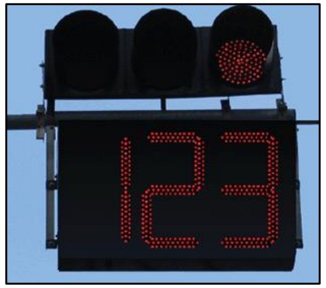
Transitions



- 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

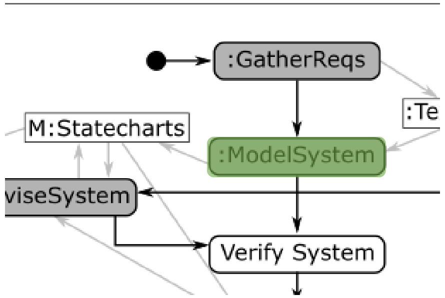
event(params) [guard] / output_action(params)

(Simulated) Plant



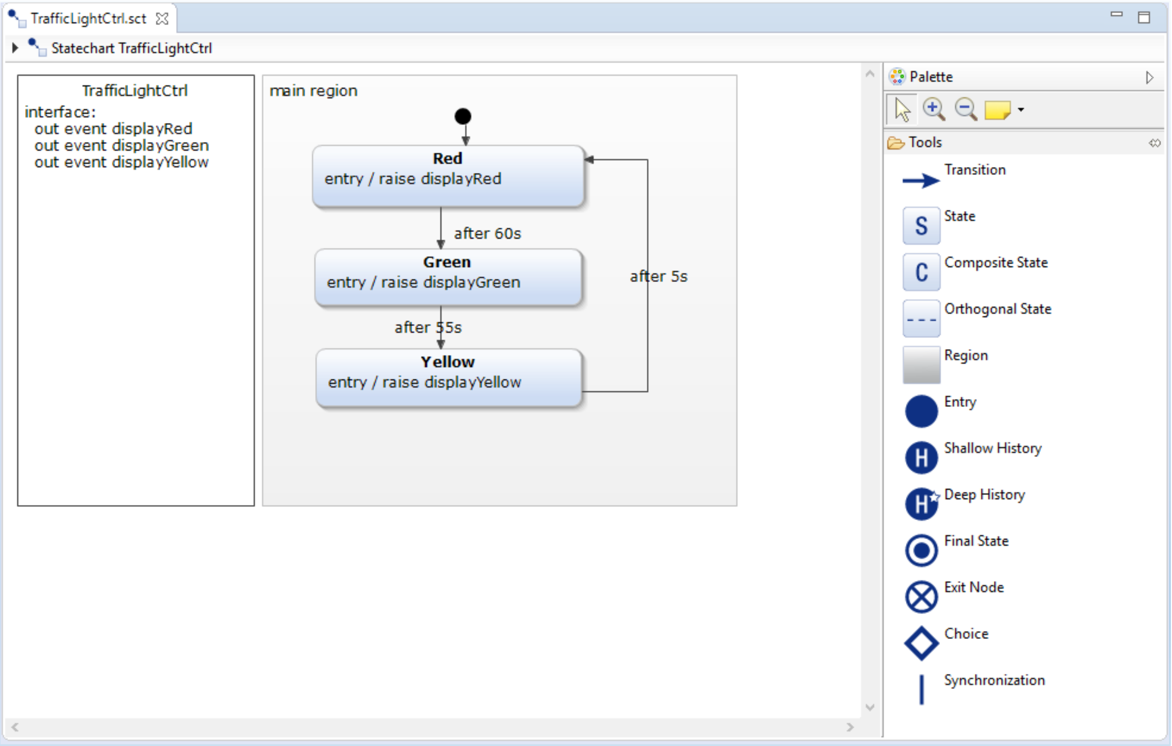
<<observe>>

Environment



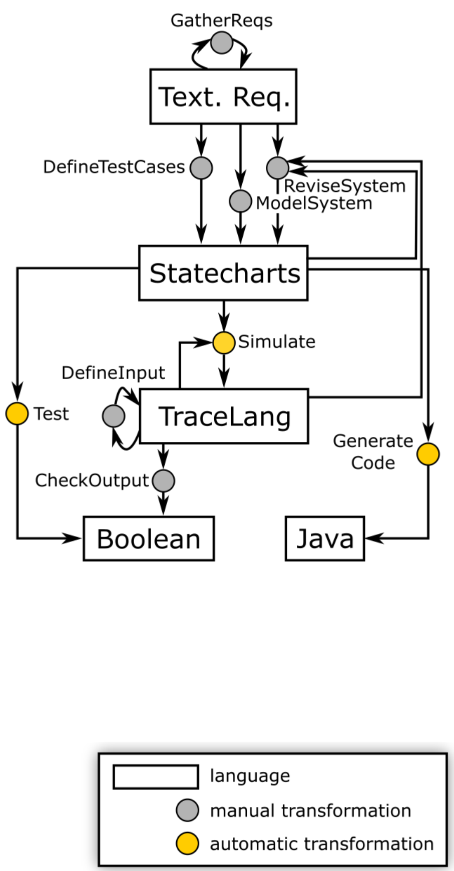
Yakindu[5]: Modelling

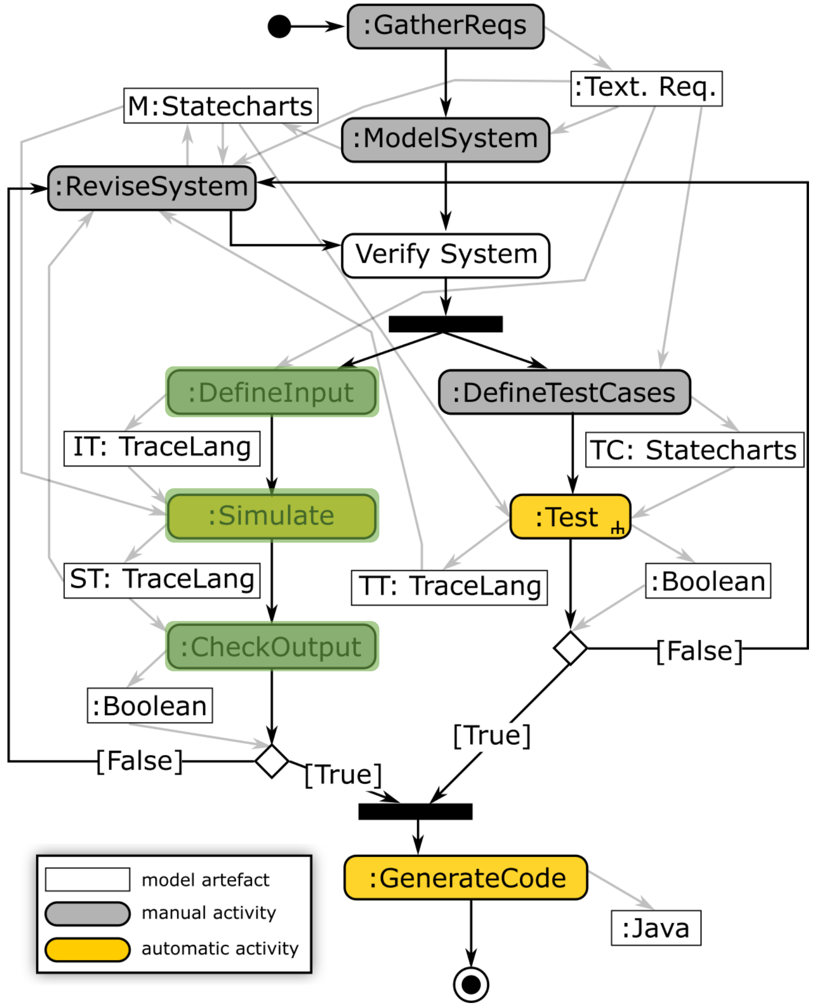
(introducing syntactic sugar: **enter actions**)



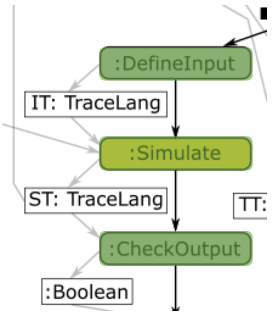
[5] https://www.itemis.com/en/yakindu/state-machine/

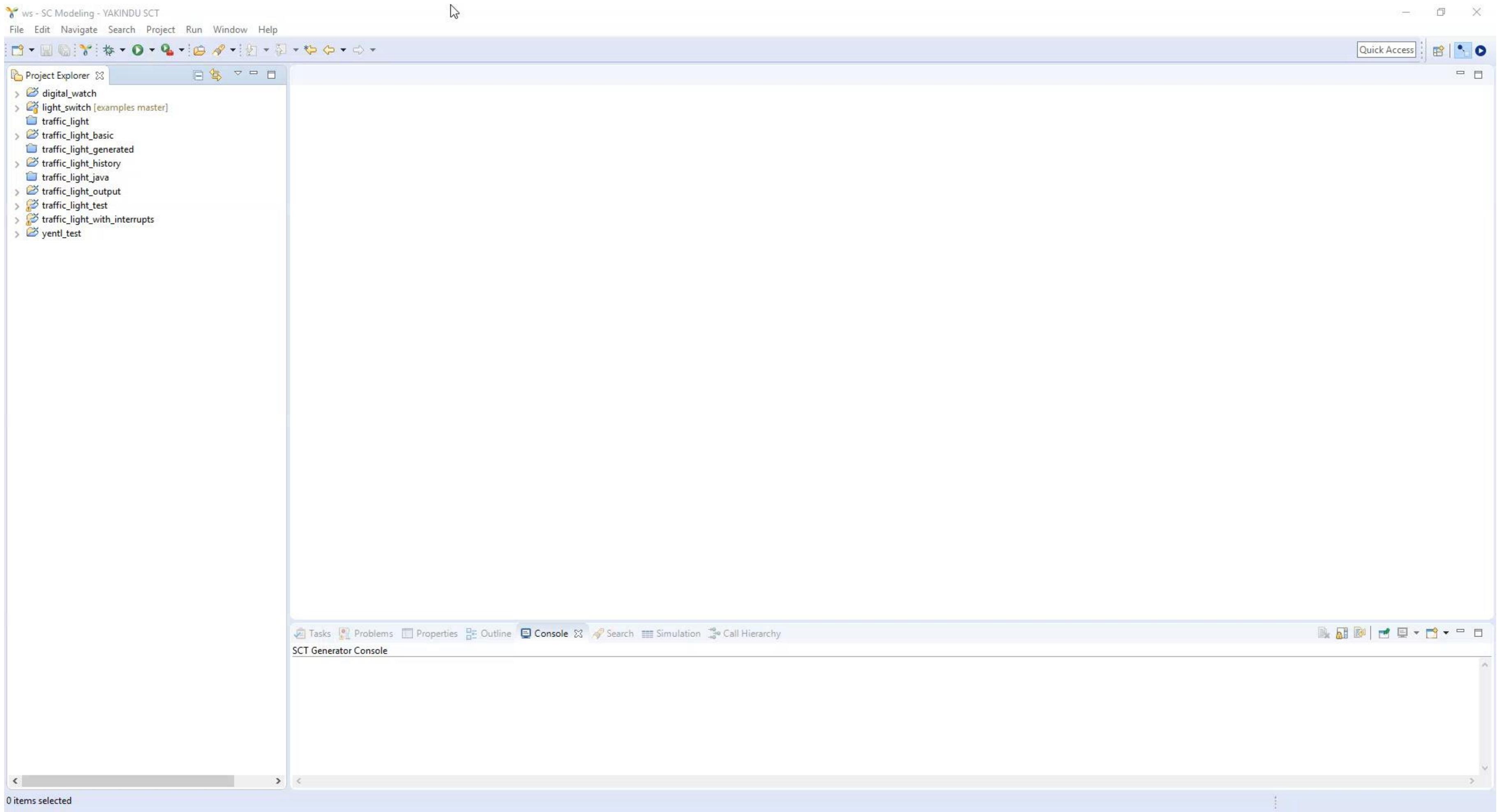
Workflow


GatherReqs
Text. Req.
DefineTestCases
ReviseSystem
ModelSystem
Statecharts
Simulate
DefineInput
Test
TraceLang
Generate Code
CheckOutput
Boolean
Java
language
manual transformation
automatic transformation

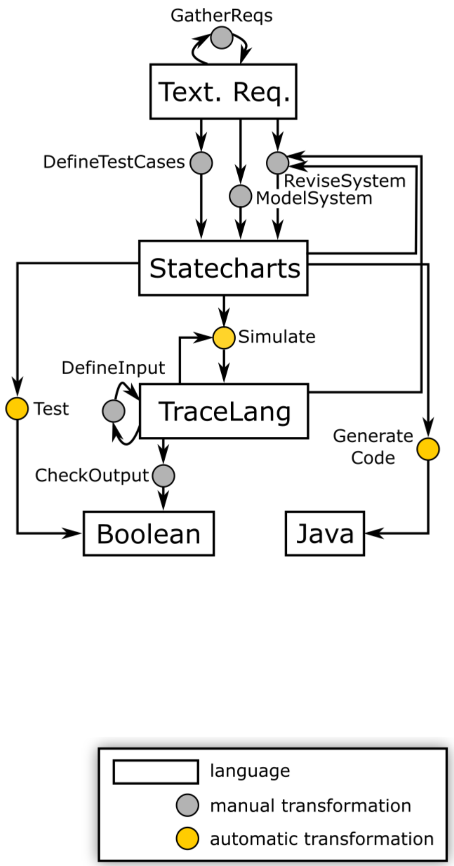

:GatherReqs
:Text. Req.
M:Statecharts
:ModelSystem
:ReviseSystem
Verify System
:DefineInput
:DefineTestCases
IT: TraceLang
TC: Statecharts
:Simulate
:Test
ST: TraceLang
TT: TraceLang
:Boolean
:CheckOutput
[False]
[True]
:Boolean
[False]
[True]
:GenerateCode
:Java
model artefact
manual activity
automatic activity

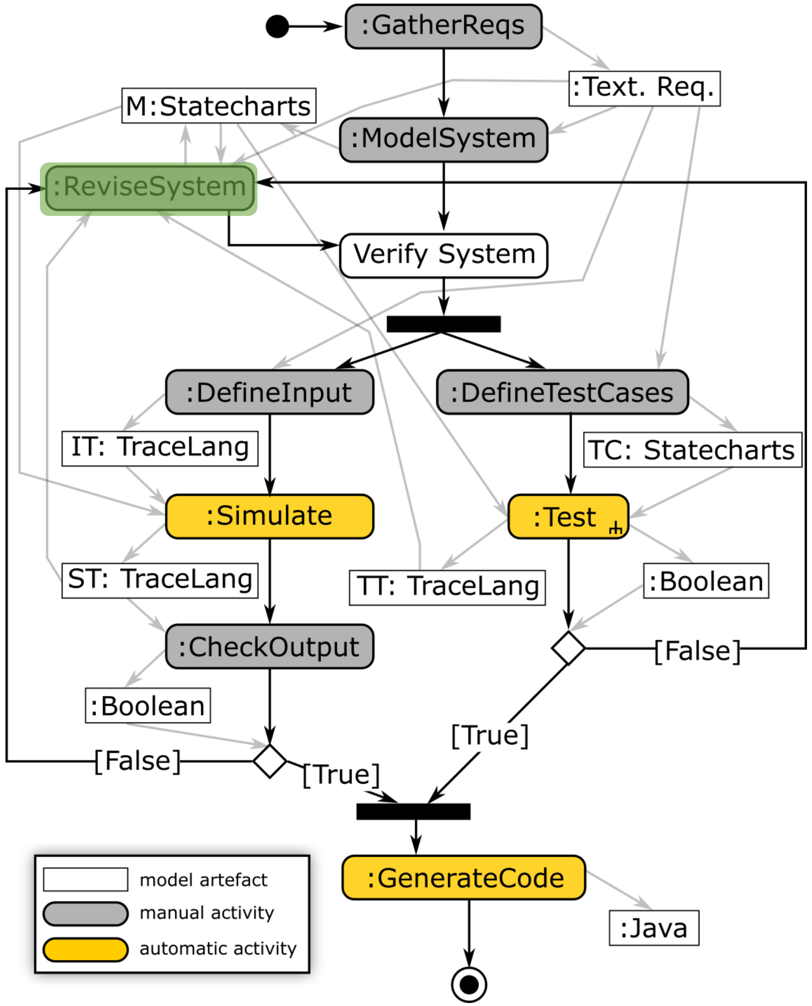
Yakindu: Simulation (Scaled Real-Time)

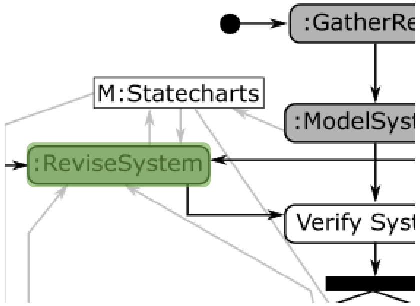




Workflow

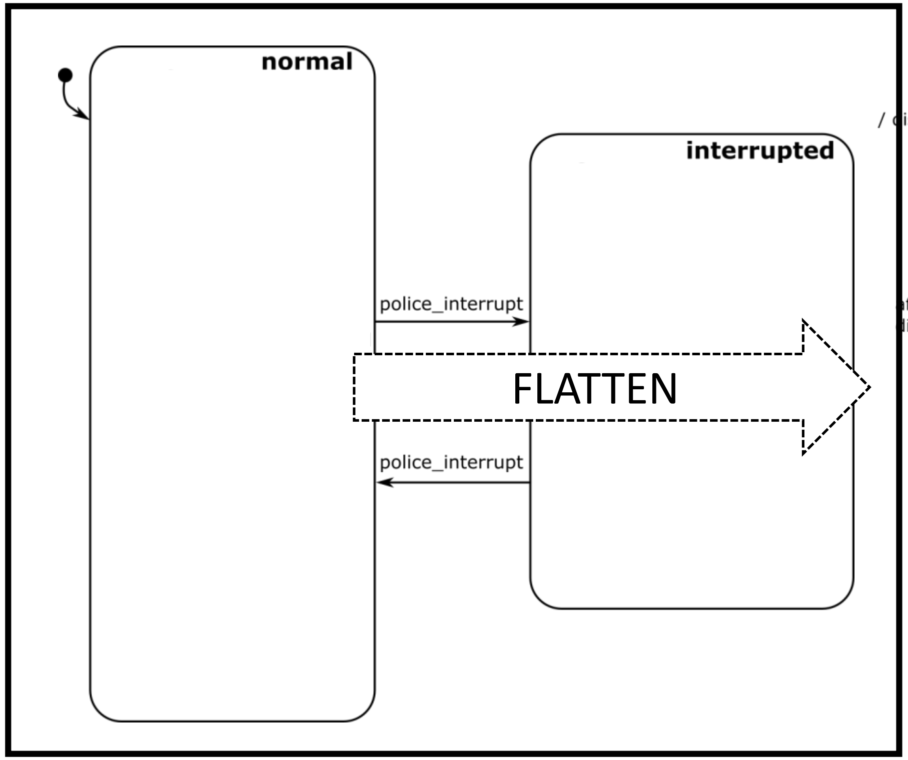

GatherReqs
Text. Req.
DefineTestCases
ReviseSystem
ModelSystem
Statecharts
Simulate
DefineInput
Test
TraceLang
Generate Code
CheckOutput
Boolean
Java
language
manual transformation
automatic transformation


:GatherReqs
:Text. Req.
M:Statecharts
:ModelSystem
:ReviseSystem
Verify System
:DefineInput
:DefineTestCases
IT: TraceLang
TC: Statecharts
:Simulate
:Test
ST: TraceLang
TT: TraceLang
:Boolean
:CheckOutput
[False]
:Boolean
[True]
[False]
[True]
:GenerateCode
:Java
model artefact
manual activity
automatic activity

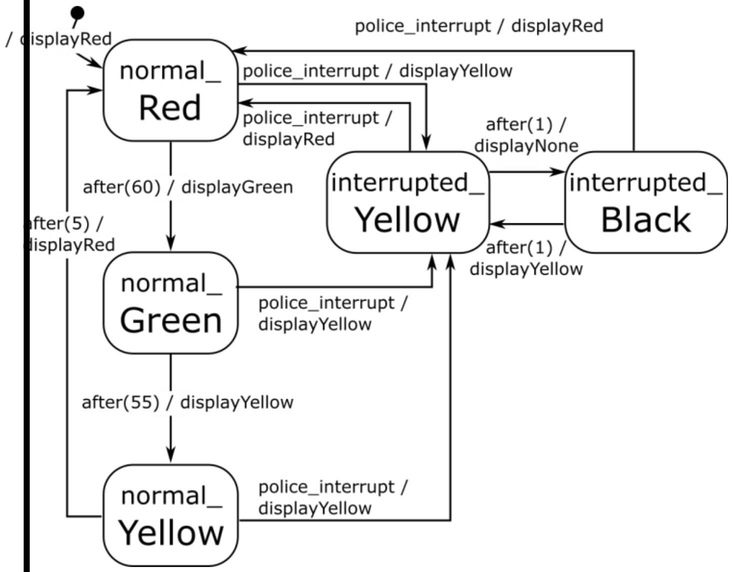


Hierarchy

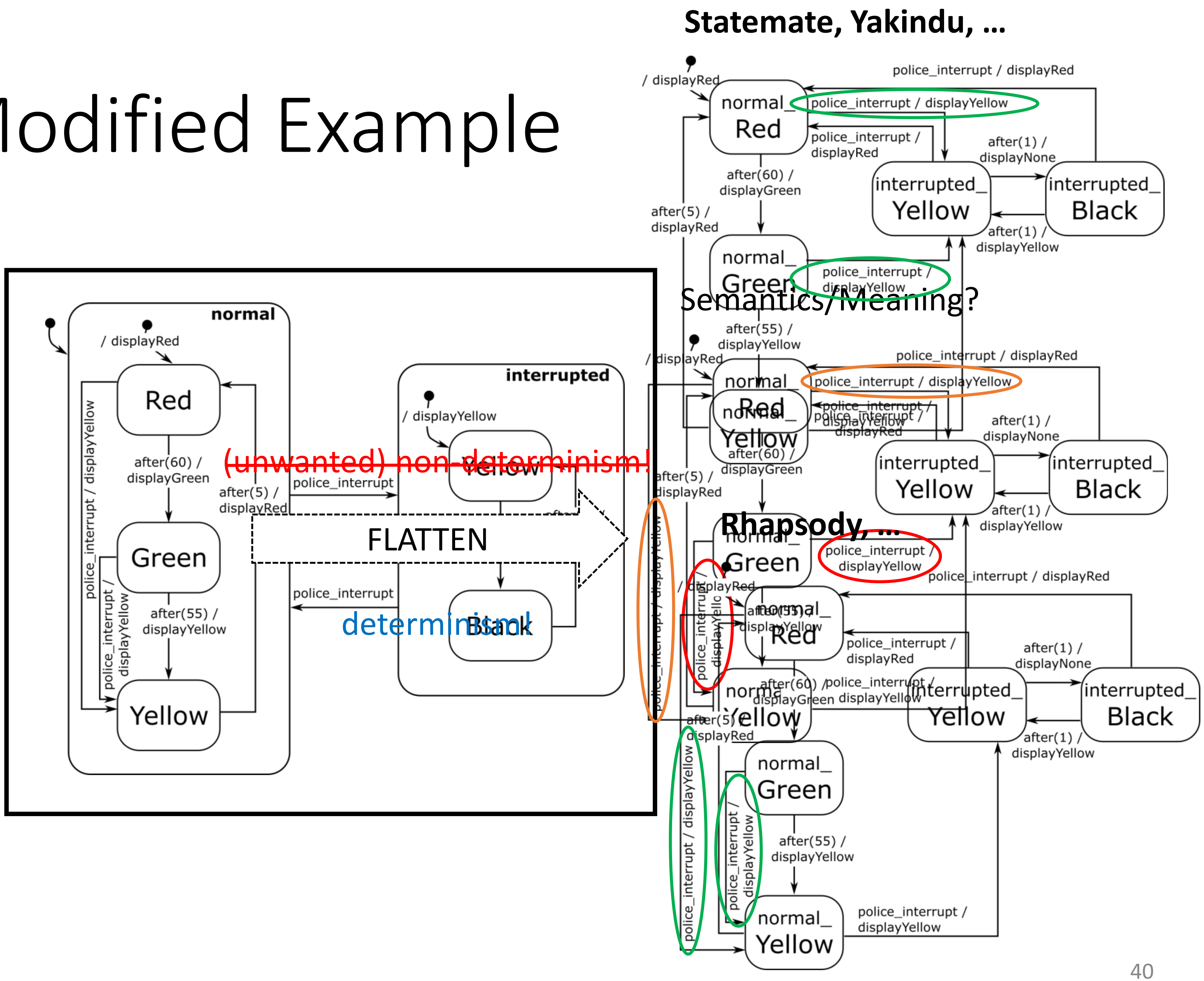
- R6: police can interrupt autonomous operation
  - Result = blinking yellow light (on -> 1s, off -> 1s)
- R7: police can resume an interrupted traffic light



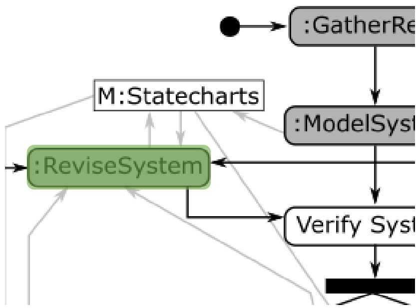
Semantics/Meaning?

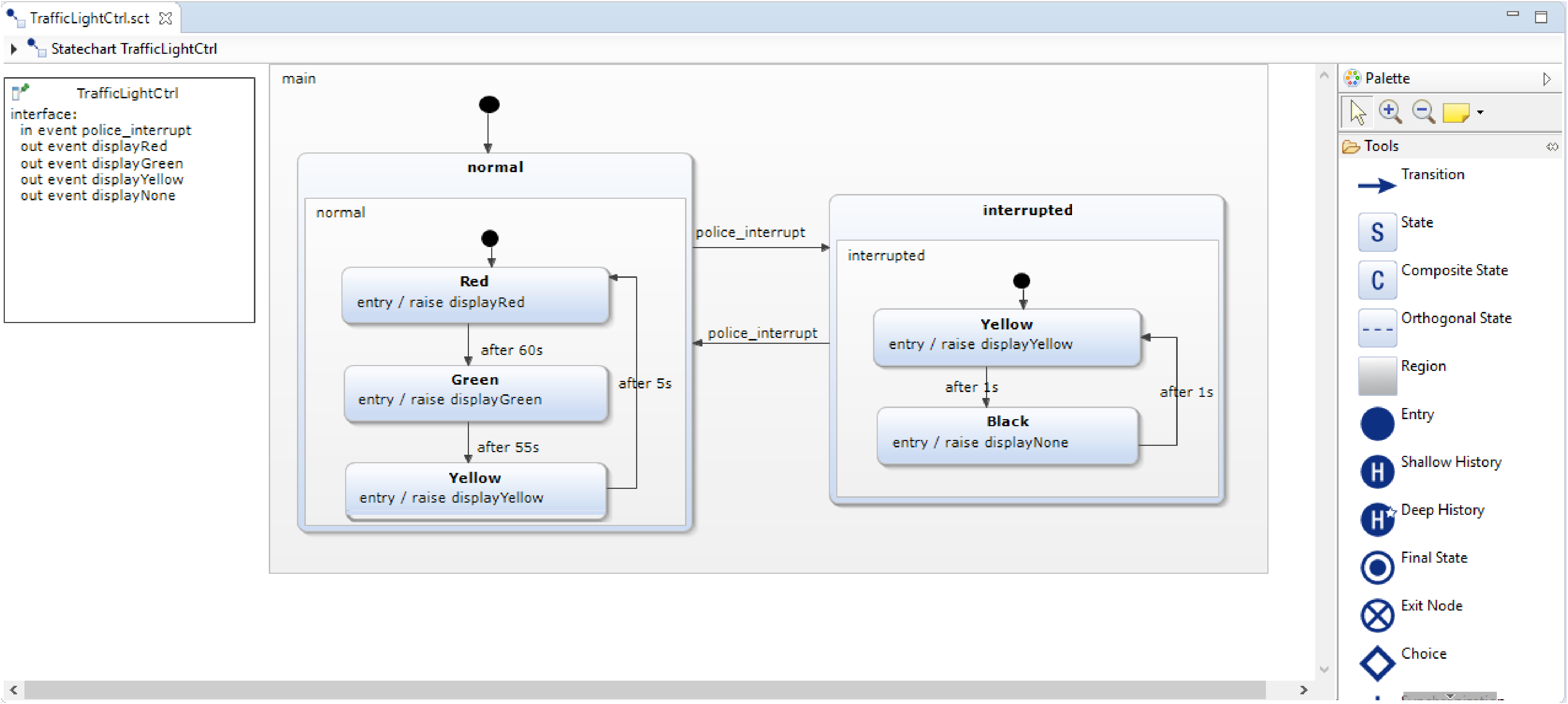


Hierarchy: Modified Example

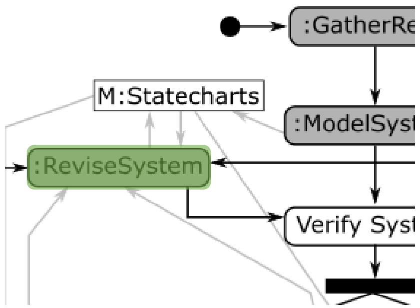


Yakindu: Hierarchy

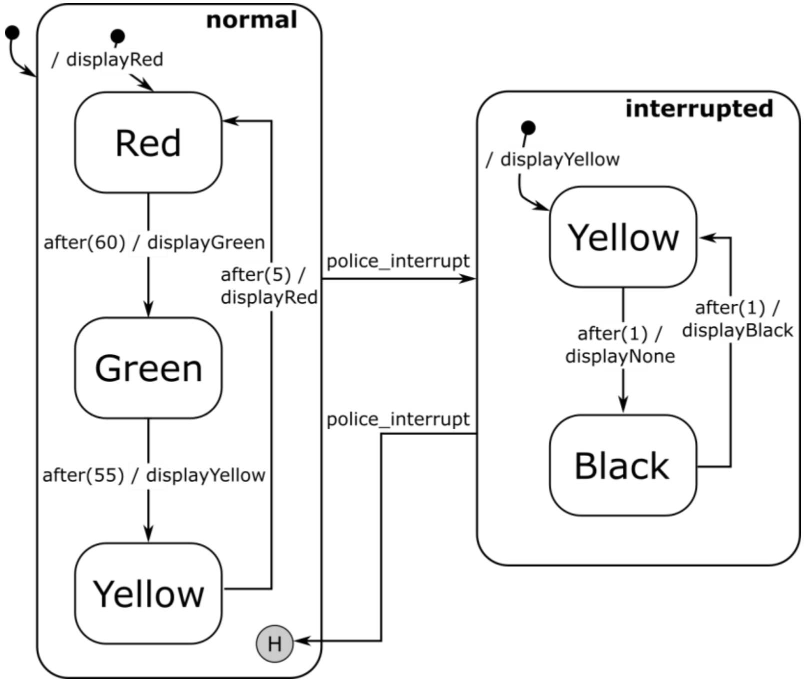




History



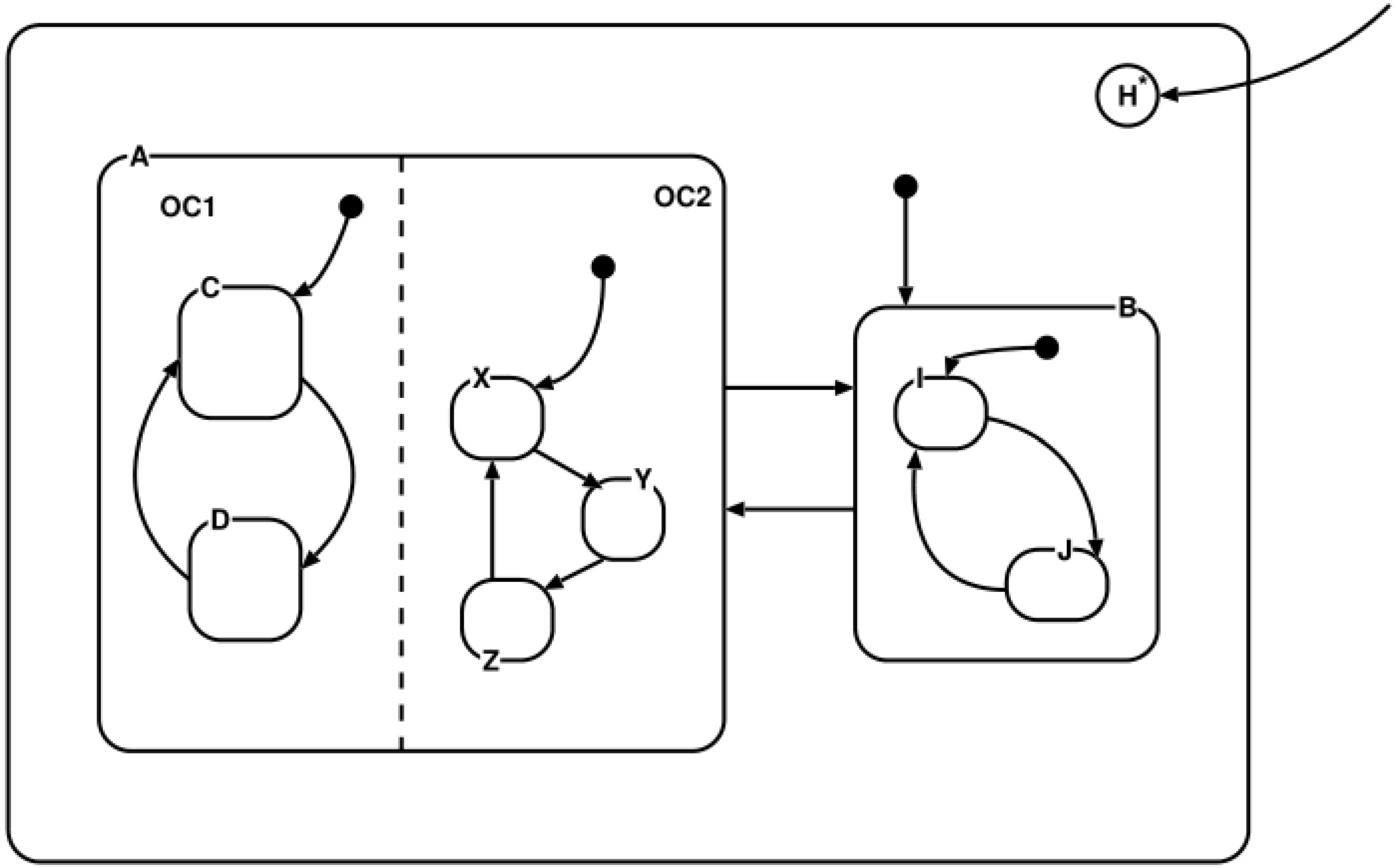
- R7: police can resume an interrupted traffic light
  - Result = light which was on at time of interrupt is turned on again



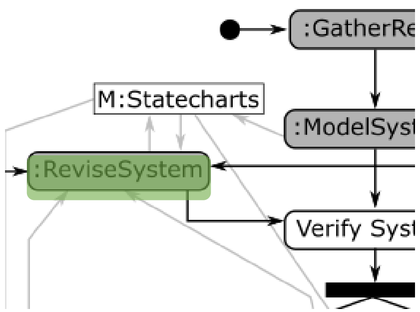
H *shallow* history

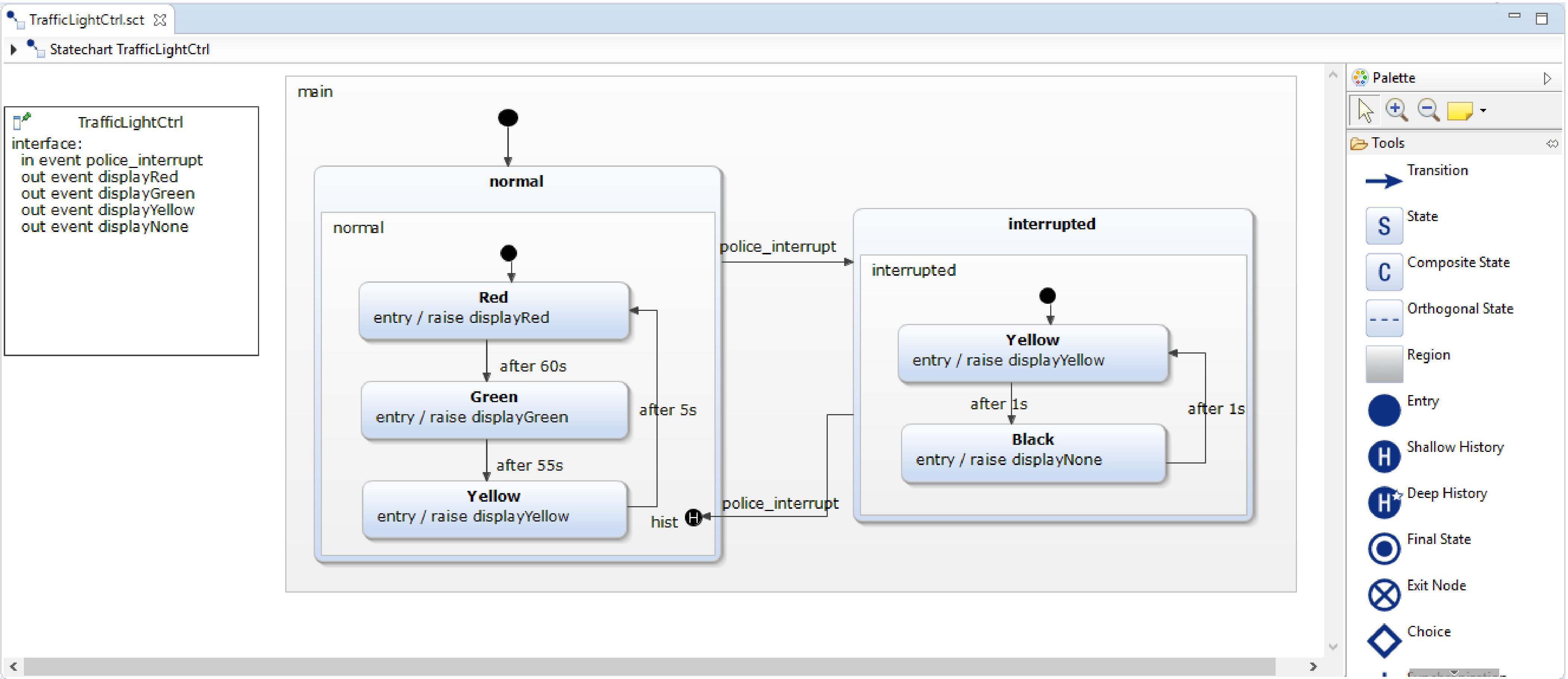
H* *deep* history

Deep History

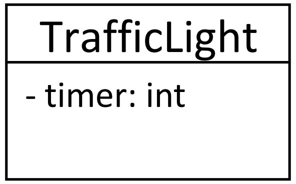


Yakindu: History

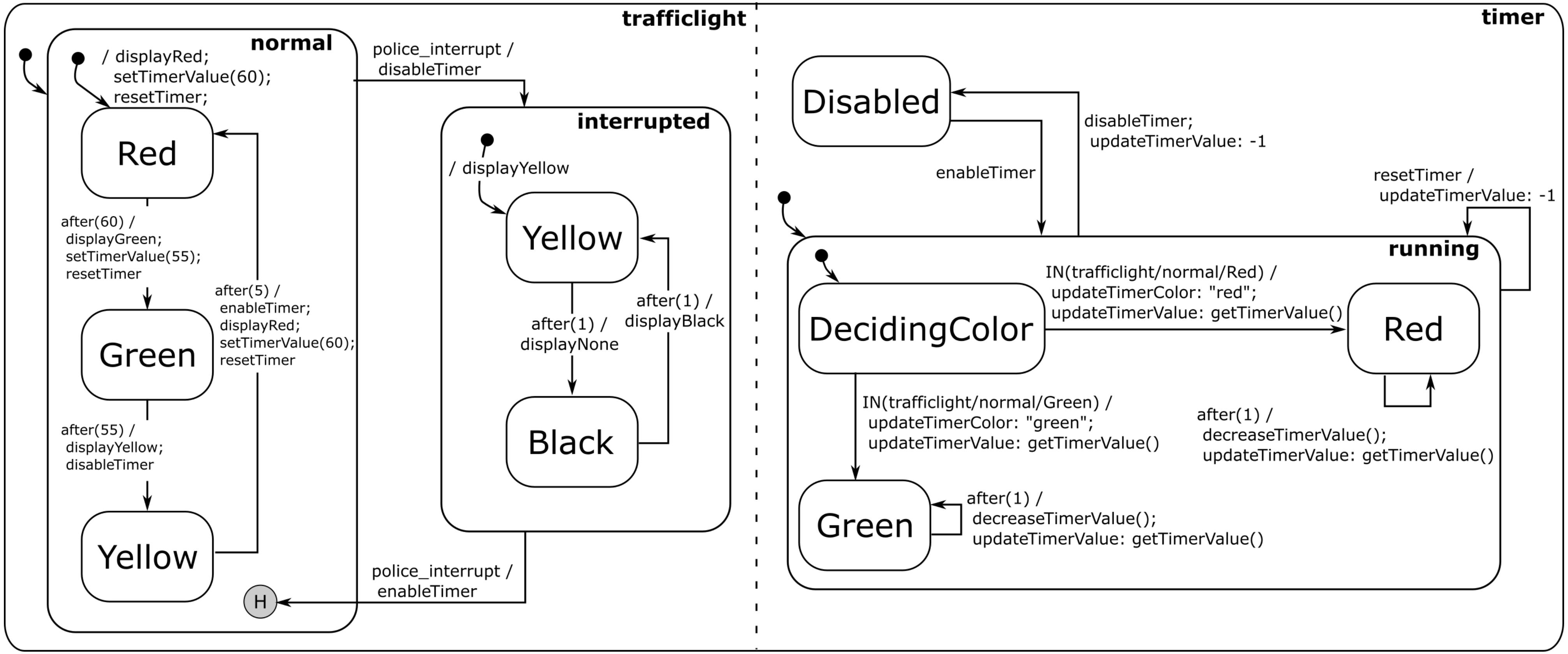




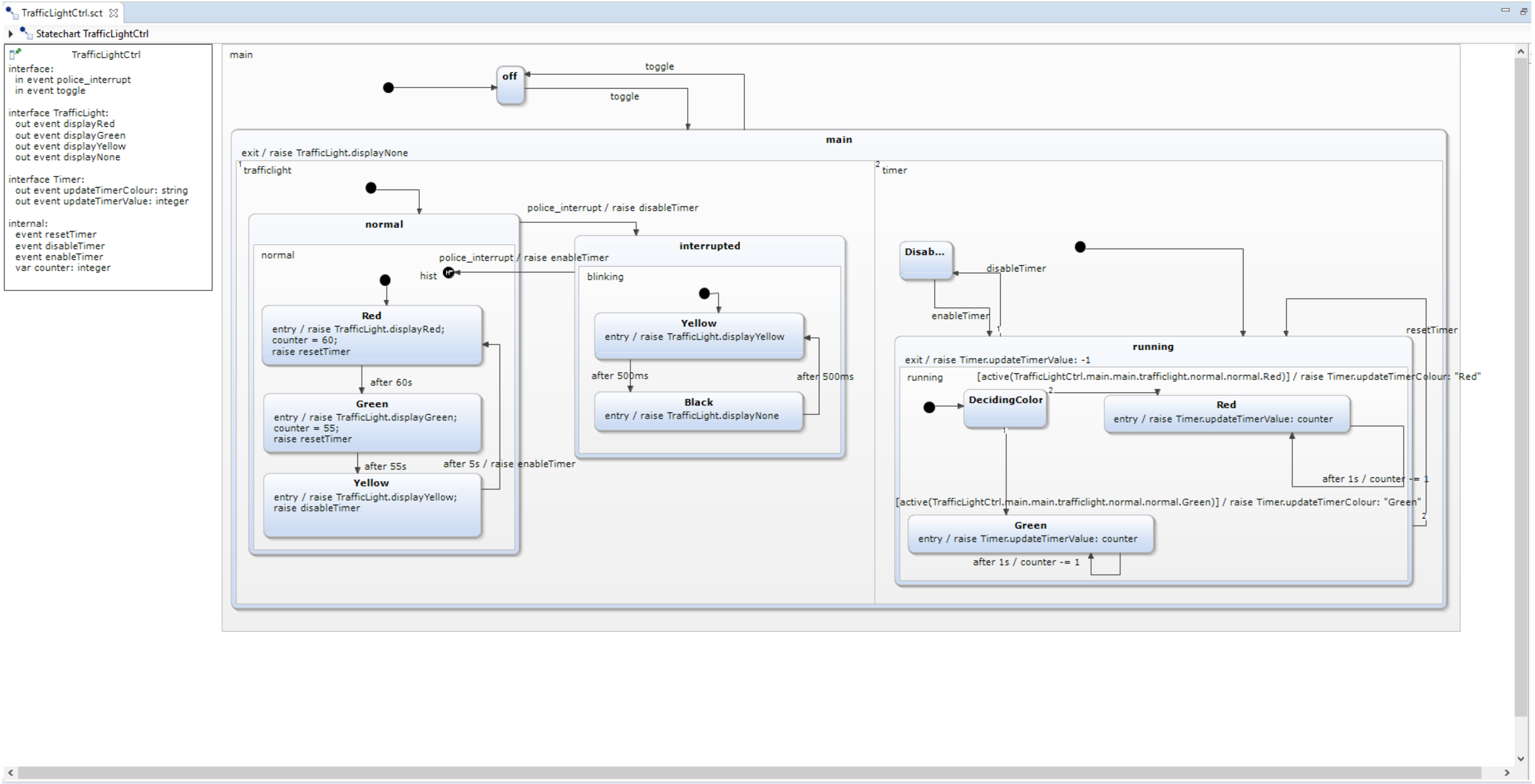
Concurrency



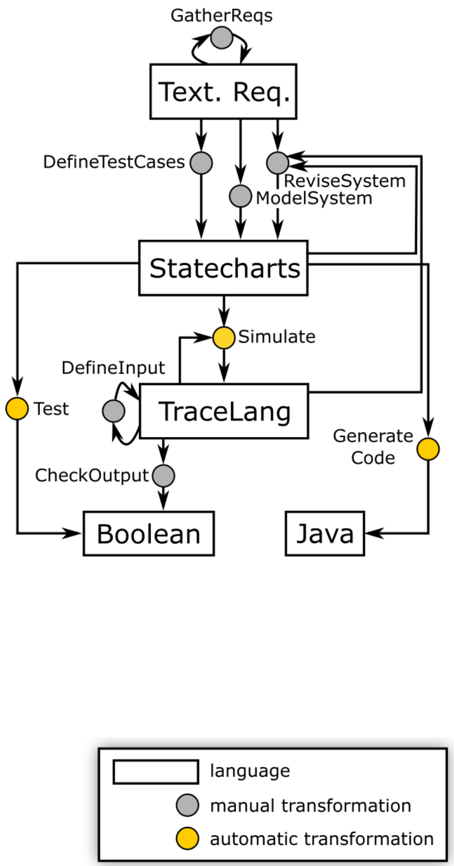
- R8: a timer displays the remaining time while the light is red or green; this timer decreases and displays its value every second. The colour of the timer reflects the colour of the traffic light.



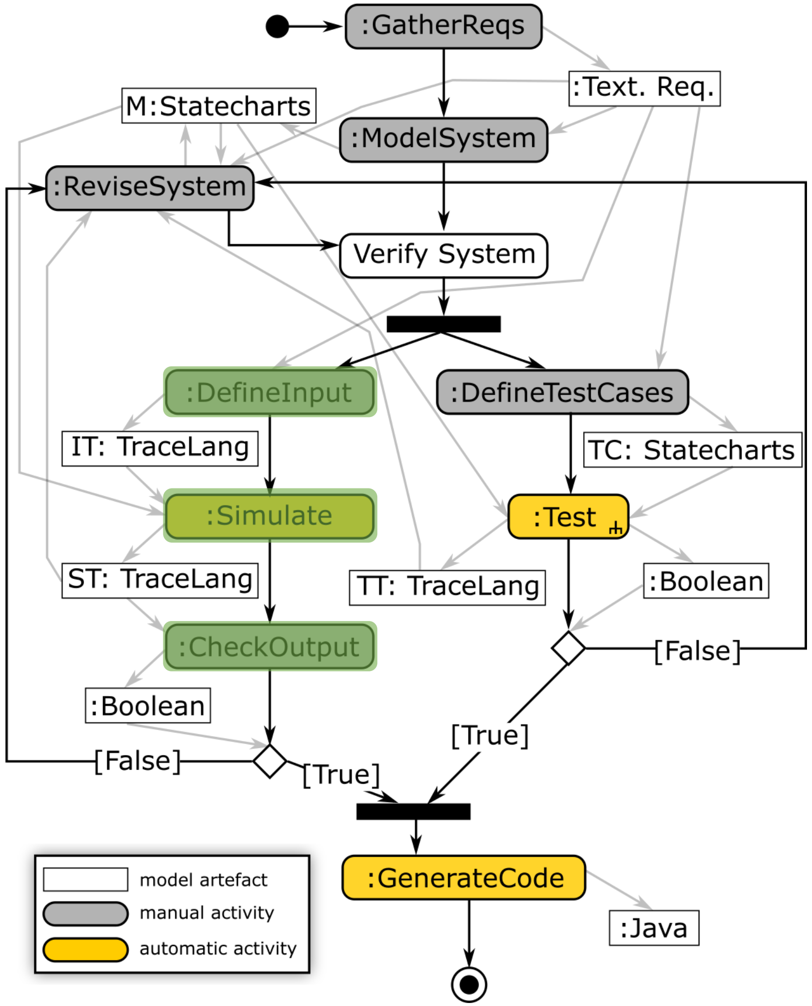
Yakindu: Concurrency



Workflow

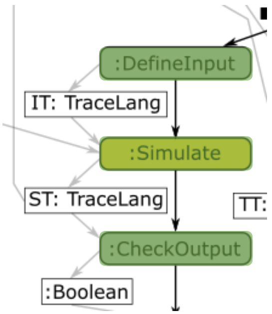


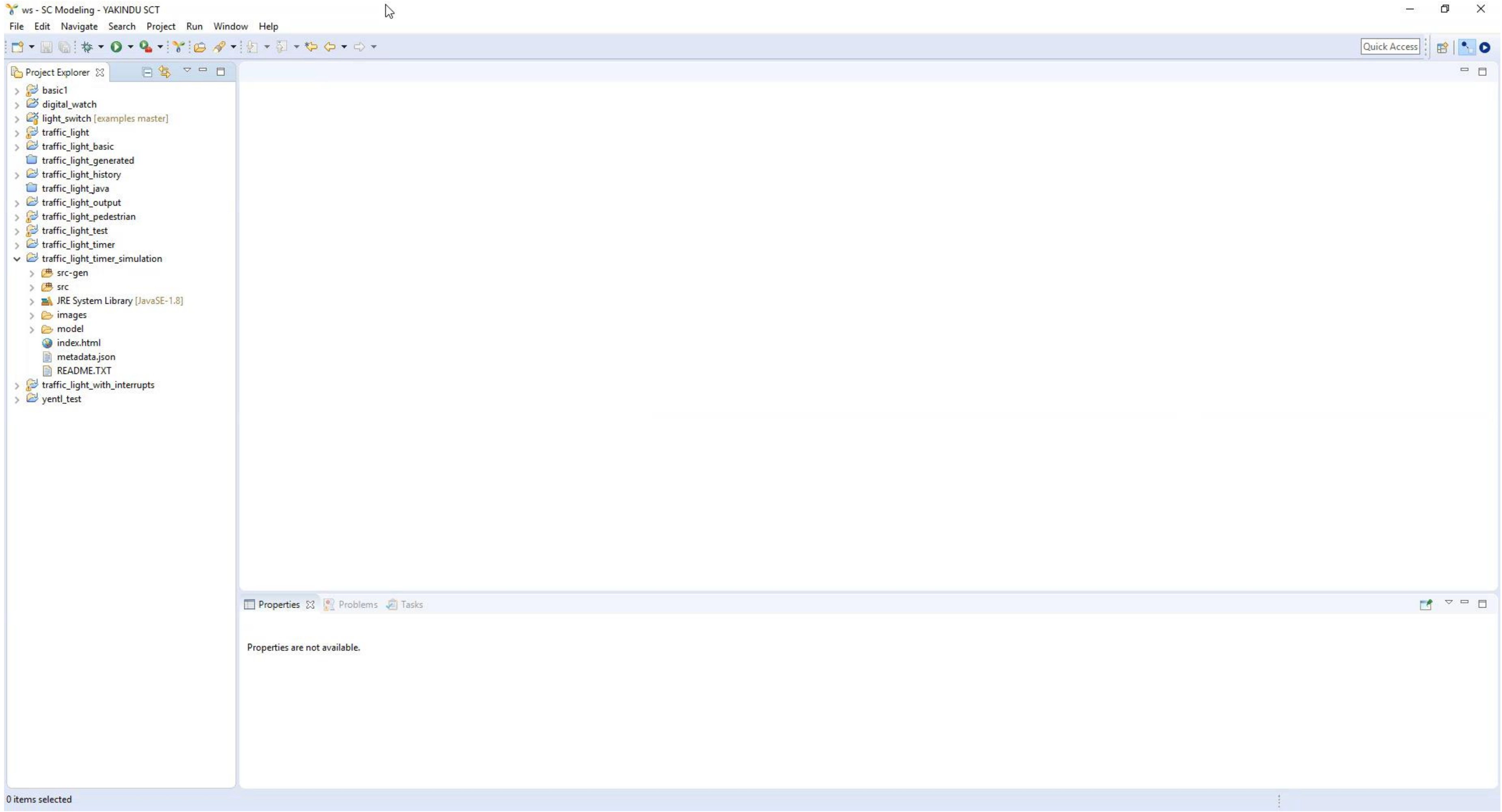
GatherReqs
Text. Req.
DefineTestCases
ReviseSystem
ModelSystem
Statecharts
Simulate
DefineInput
Test
TraceLang
Generate Code
CheckOutput
Boolean
Java
language
manual transformation
automatic transformation



:GatherReqs
:Text. Req.
M:Statecharts
:ModelSystem
:ReviseSystem
Verify System
:DefineInput
:DefineTestCases
IT: TraceLang
TC: Statecharts
:Simulate
:Test
ST: TraceLang
TT: TraceLang
:Boolean
:CheckOutput
[False]
:Boolean
[True]
[False]
[True]
:GenerateCode
:Java
model artefact
manual activity
automatic activity

Yakindu: Simulation (Scaled Real-Time)

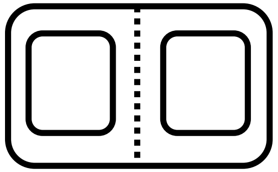




Statechart Semantics: Initialization

```
init(sc):
   targetStates =
      getEffectiveTargetStates(getDefaultState(sc))
   for target in targetStates:
      enter(target)
```

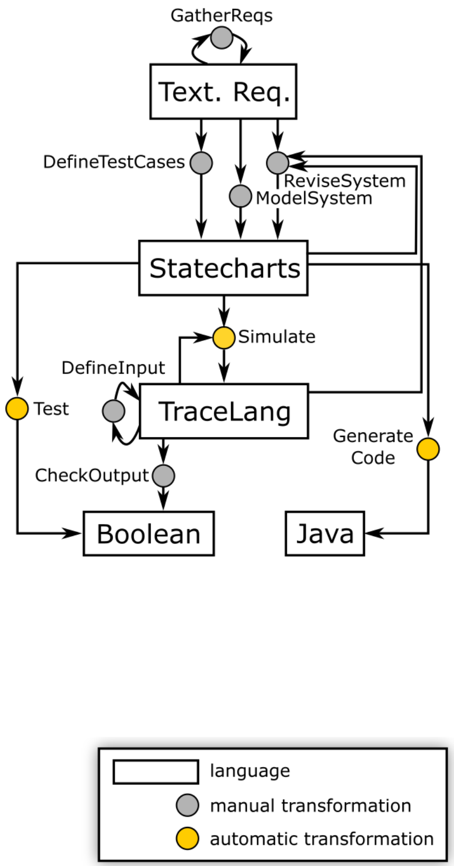
Statecharts Semantics: “Main Loop”

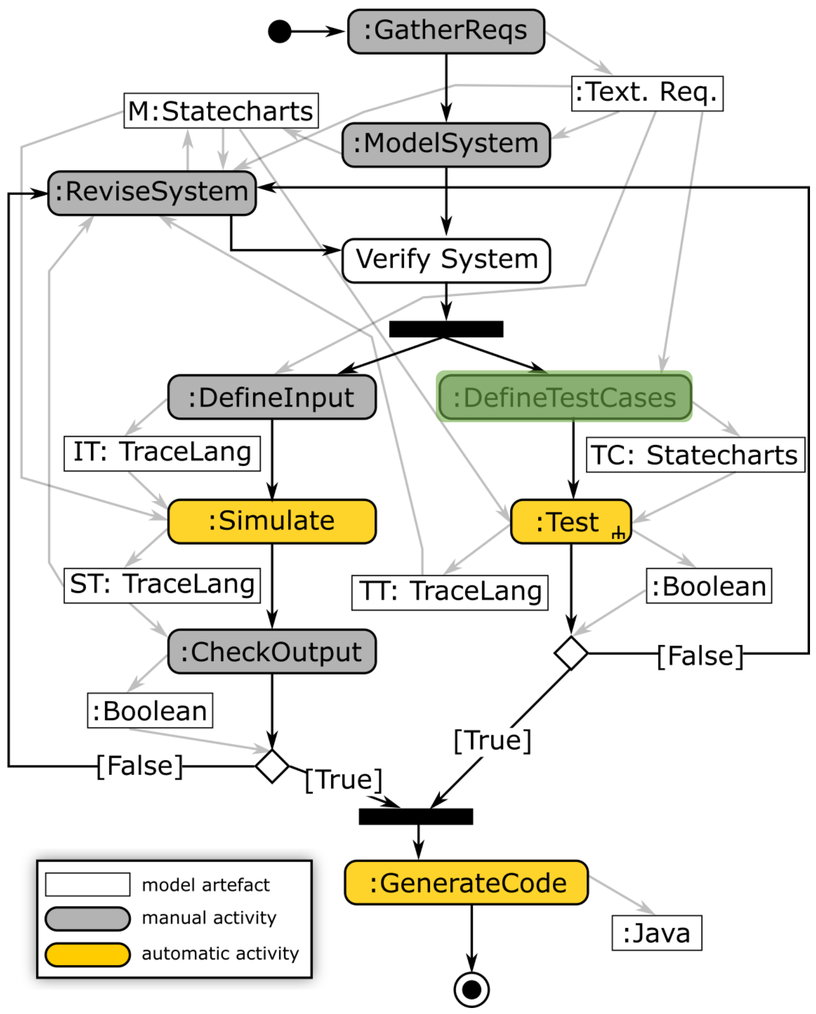


```
while True:
   for all concurrent regions:
      candidates =
findEnabledTransitions(getEnabledEvents(),
                                   getCurrentState())
      removeConflicts(candidates)
      execute(chooseOne(candidates))
```

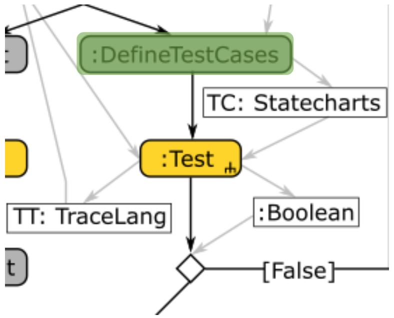
1. Find LCA
2. Leave states up the hierarchy
3. Execute action *a*
4. Enter states down the hierarchy (getEffectiveTargetStates())

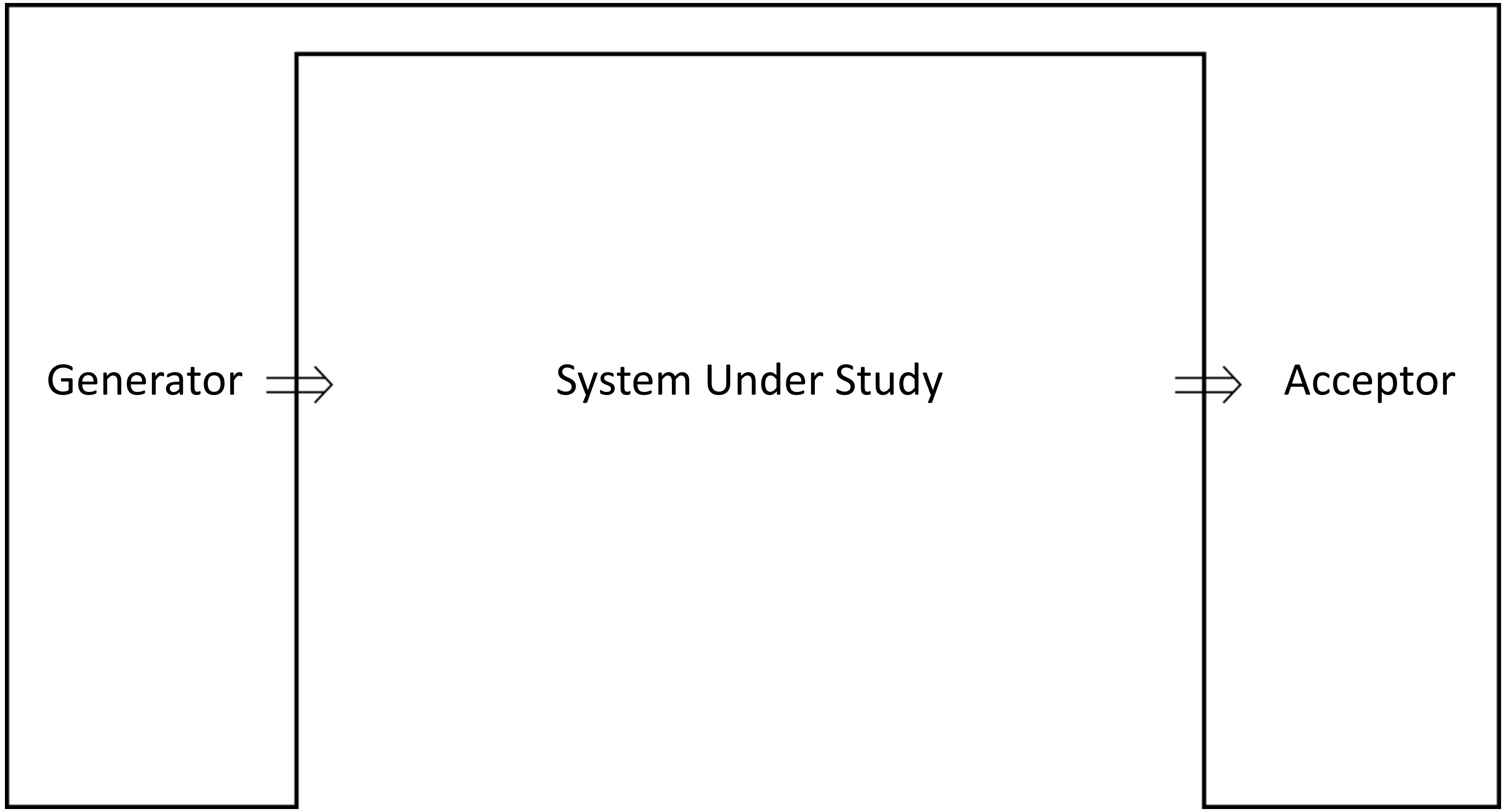
Workflow


GatherReqs
Text. Req.
DefineTestCases
ReviseSystem
ModelSystem
Statecharts
Simulate
DefineInput
Test
TraceLang
Generate Code
CheckOutput
Boolean
Java
language
manual transformation
automatic transformation


:GatherReqs
:Text. Req.
M:Statecharts
:ModelSystem
:ReviseSystem
Verify System
:DefineInput
:DefineTestCases
IT: TraceLang
TC: Statecharts
:Simulate
:Test
ST: TraceLang
TT: TraceLang
:Boolean
:CheckOutput
[False]
:Boolean
[True]
[False]
[True]
:GenerateCode
:Java
model artefact
manual activity
automatic activity

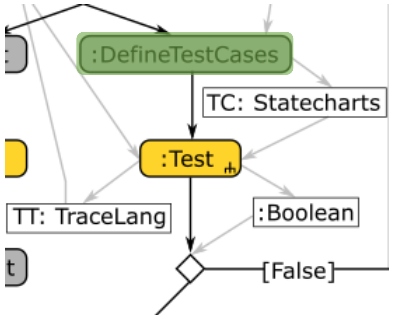
Statecharts Testing

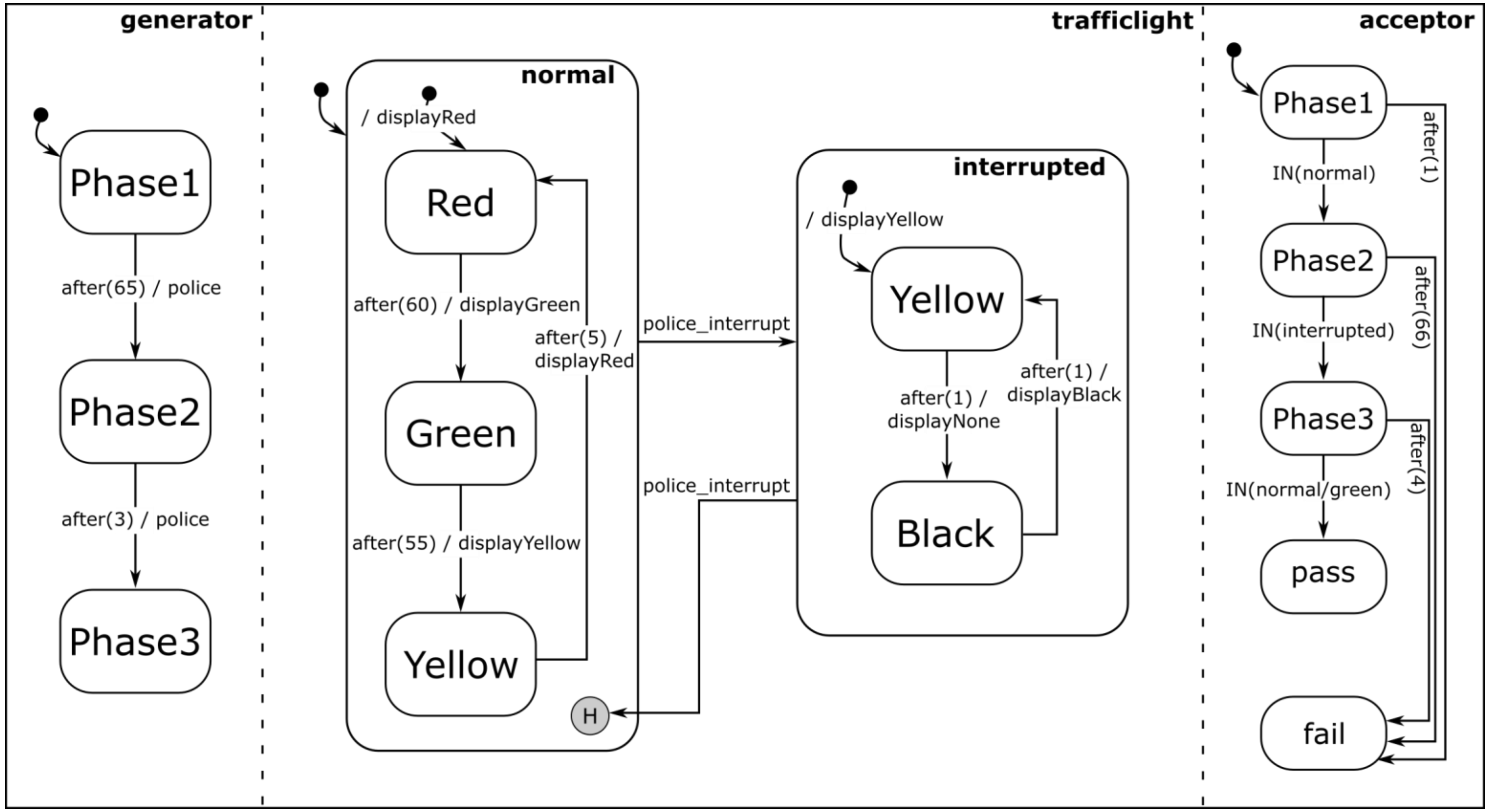




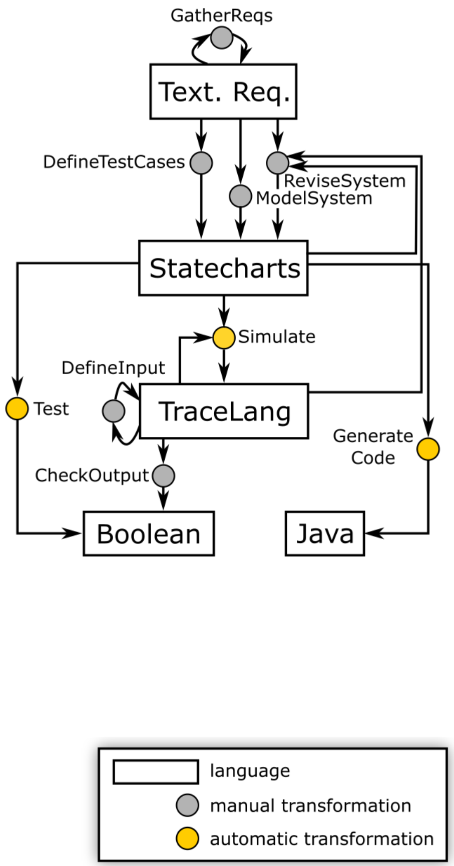
[6] Zeigler BP. Theory of modelling and simulation. New York: Wiley-Interscience, 1976.
[7] Mamadou K. Traoré, Alexandre Muzy, Capturing the dual relationship between simulation models and their context, Simulation Modelling Practice and Theory, Volume 14, Issue 2, February 2006, Pages 126-142

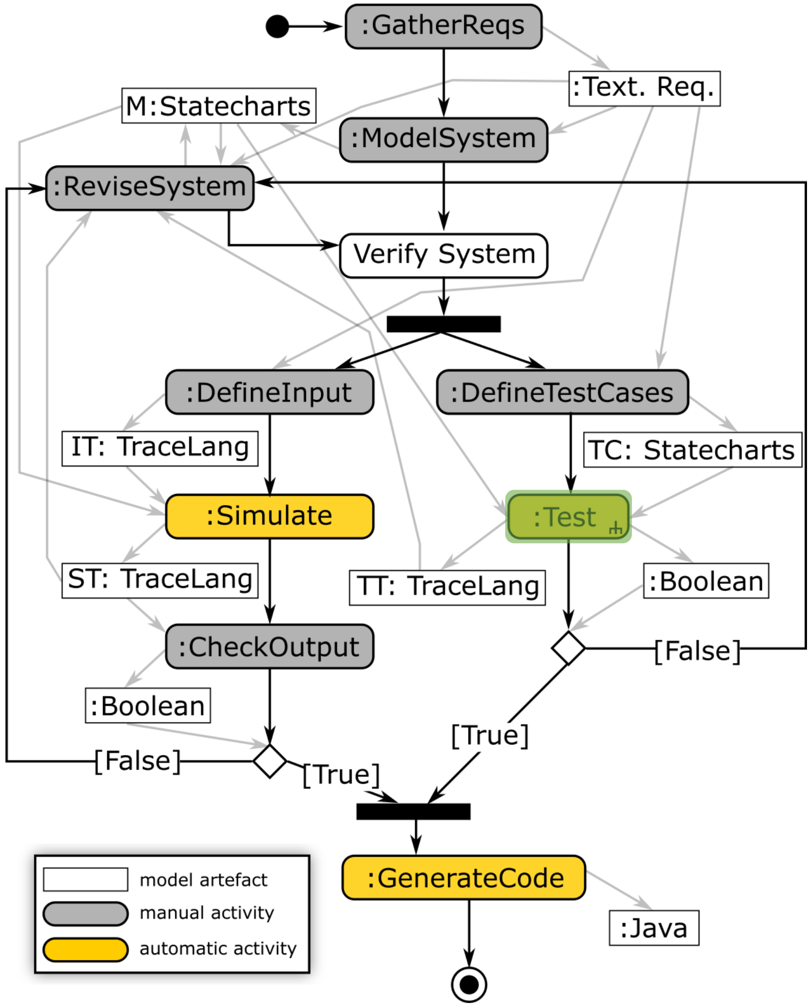
Orthogonal Components (White-Box)



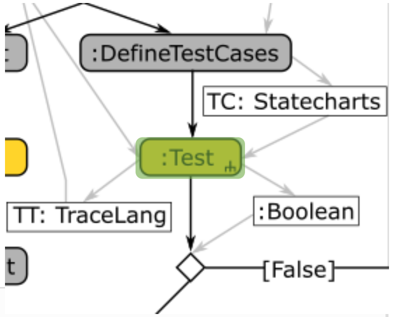


Workflow

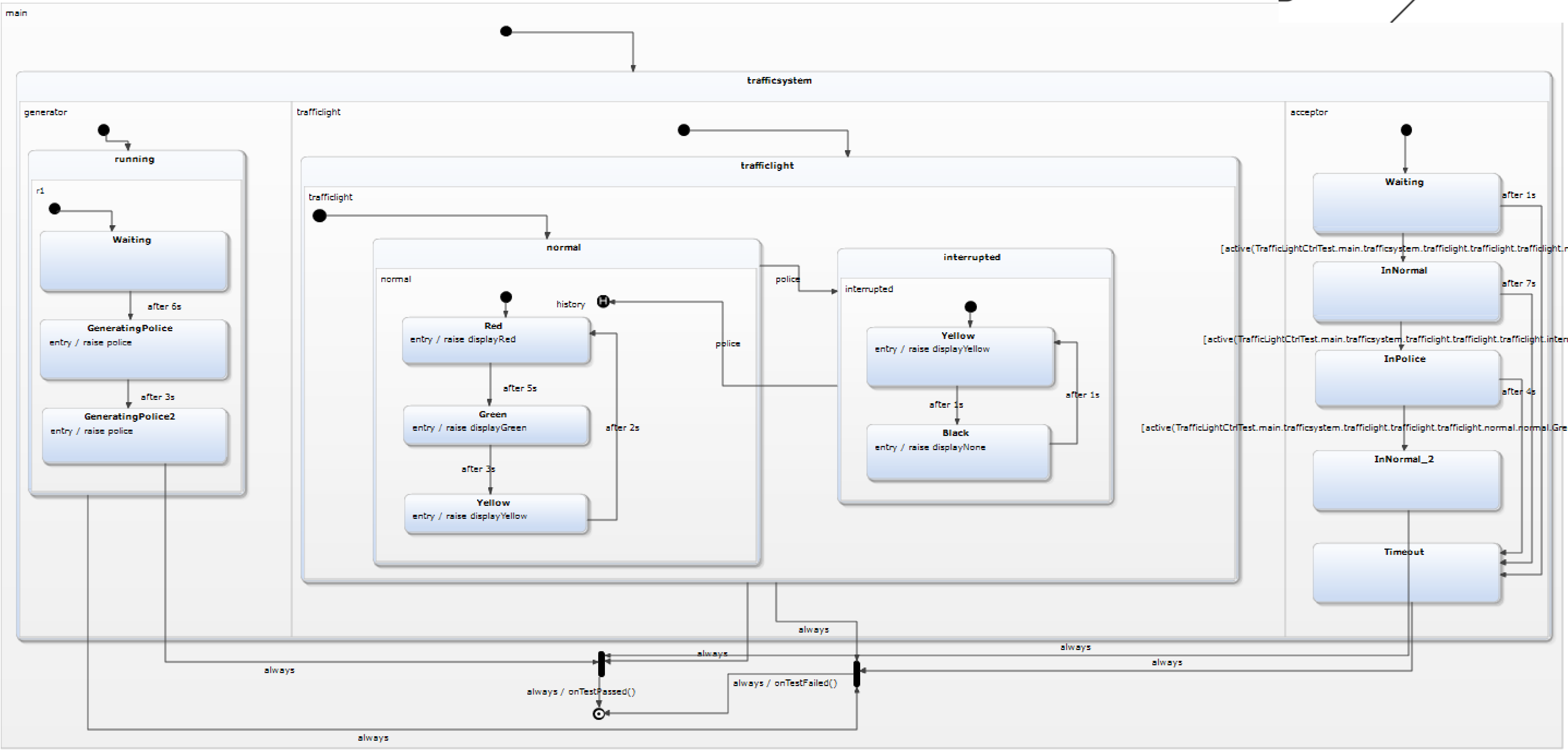

GatherReqs
Text. Req.
DefineTestCases
ReviseSystem
ModelSystem
Statecharts
Simulate
DefineInput
Test
TraceLang
Generate Code
CheckOutput
Boolean
Java
language
manual transformation
automatic transformation


:GatherReqs
:Text. Req.
M:Statecharts
:ModelSystem
:ReviseSystem
Verify System
:DefineInput
:DefineTestCases
IT: TraceLang
TC: Statecharts
:Simulate
:Test
ST: TraceLang
TT: TraceLang
:Boolean
:CheckOutput
[False]
:Boolean
[True]
[False]
[True]
:GenerateCode
:Java
model artefact
manual activity
automatic activity

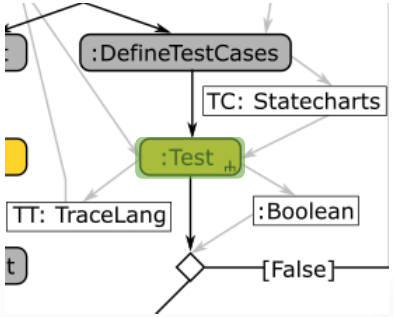
Yakindu: Testing




```
TrafficLightCtrlTest
interface:
  out event displayRed
  out event displayGreen
  out event displayYellow
  out event displayNone
  operation onTestPassed(): void
  operation onTestFailed(): void

internal:
  event police
```



Yakindu: Testing



Interface

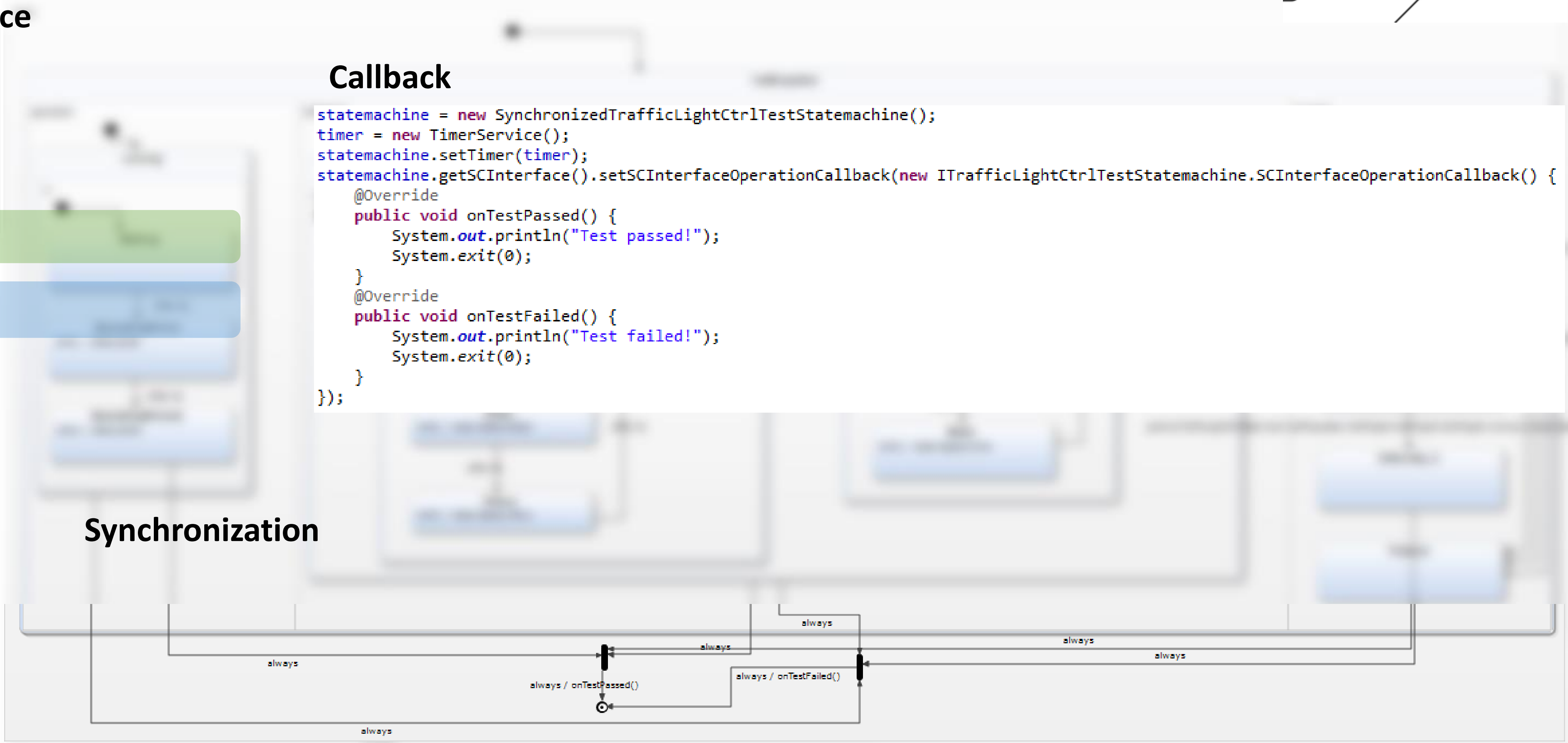
TrafficLightCtrlTest
interface:
out event displayRed
out event displayGreen
out event displayYellow
out event displayNone
operation onTestPassed(): void
operation onTestFailed(): void

internal:
event police

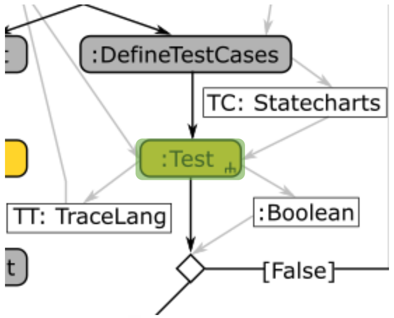
Callback

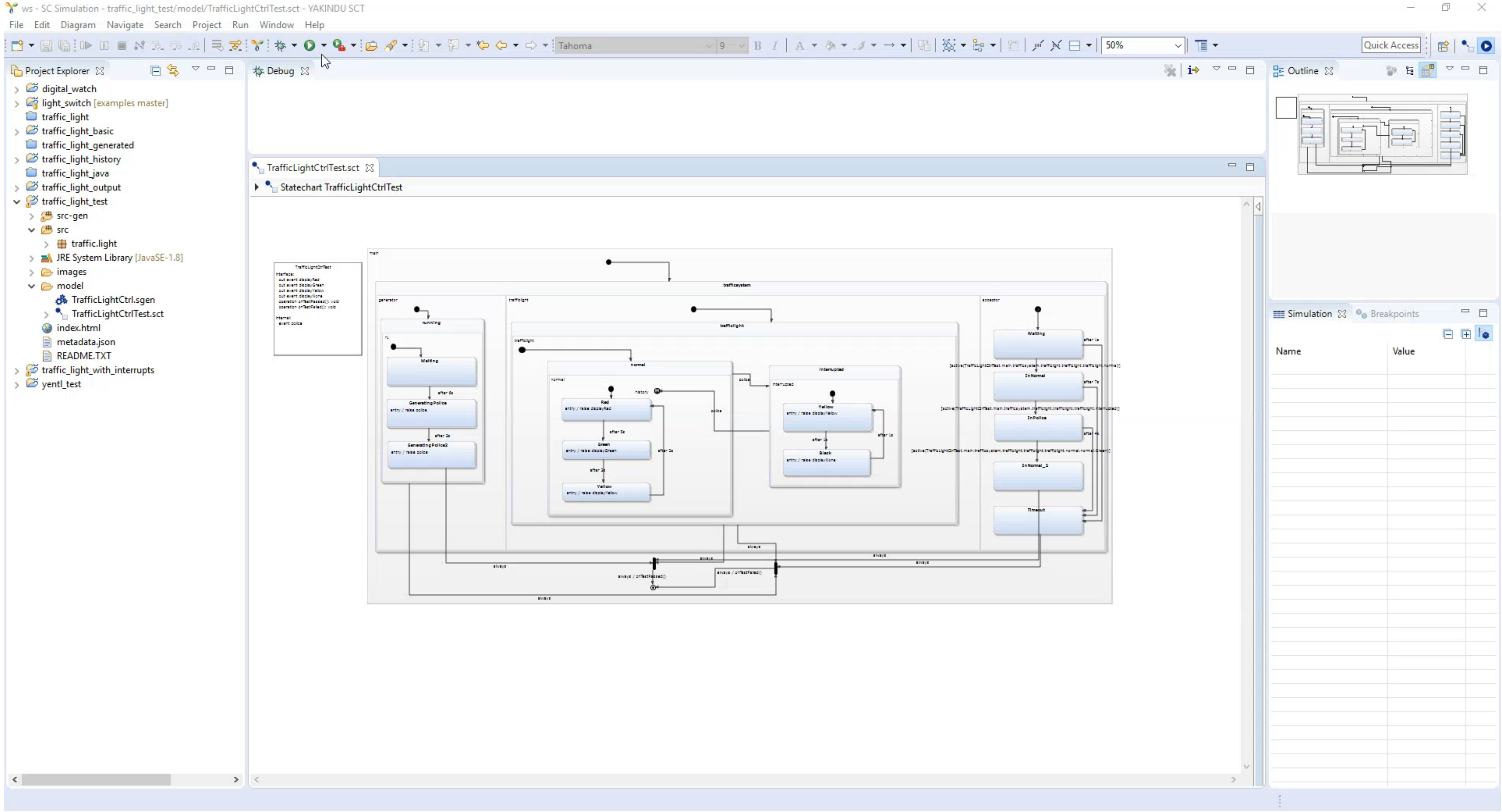
```
statemachine = new SynchronizedTrafficLightCtrlTestStatemachine();
timer = new TimerService();
statemachine.setTimer(timer);
statemachine.getSCInterface().setSCInterfaceOperationCallback(new ITrafficLightCtrlTestStatemachine.SCInterfaceOperationCallback() {
    @Override
    public void onTestPassed() {
        System.out.println("Test passed!");
        System.exit(0);
    }
    @Override
    public void onTestFailed() {
        System.out.println("Test failed!");
        System.exit(0);
    }
});
```

Synchronization

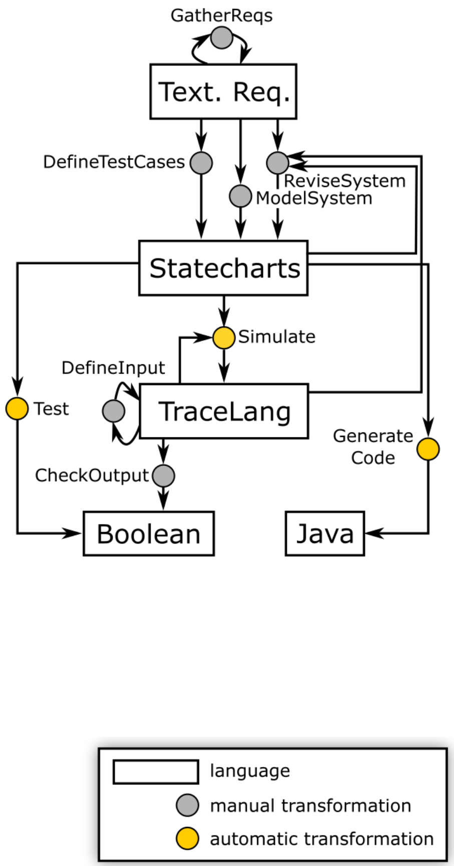


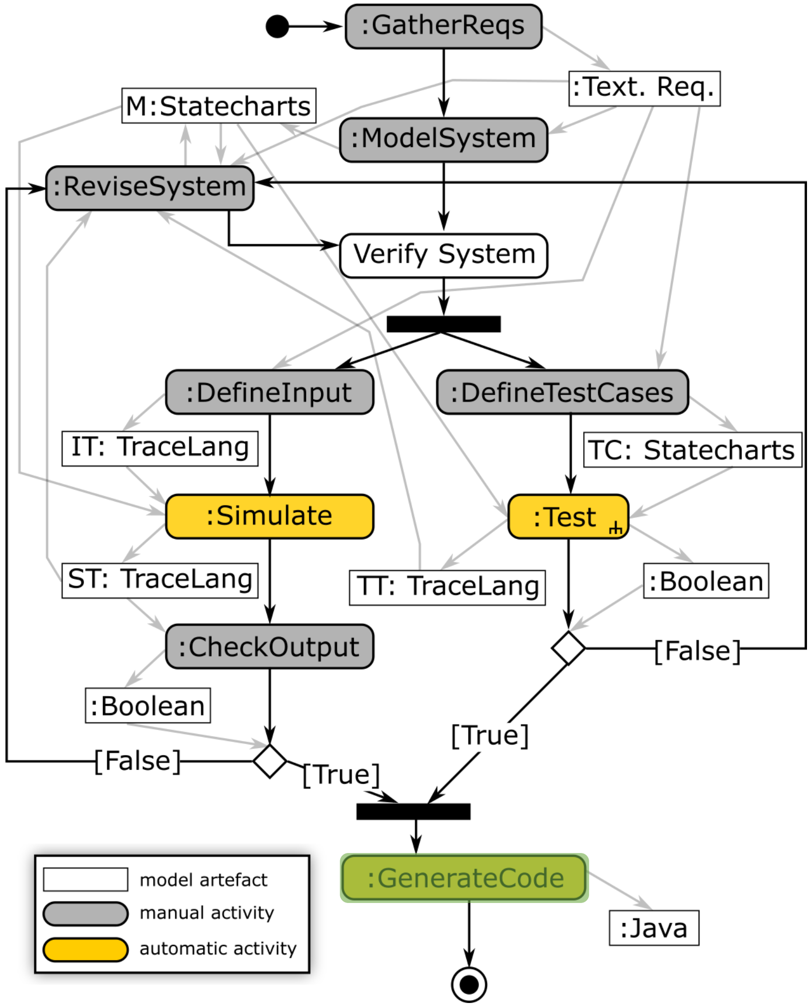
Yakindu: Testing



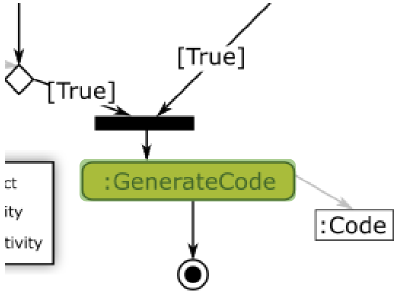


Workflow


GatherReqs
Text. Req.
DefineTestCases
ReviseSystem
ModelSystem
Statecharts
Simulate
DefineInput
Test
TraceLang
Generate Code
CheckOutput
Boolean
Java
language
manual transformation
automatic transformation

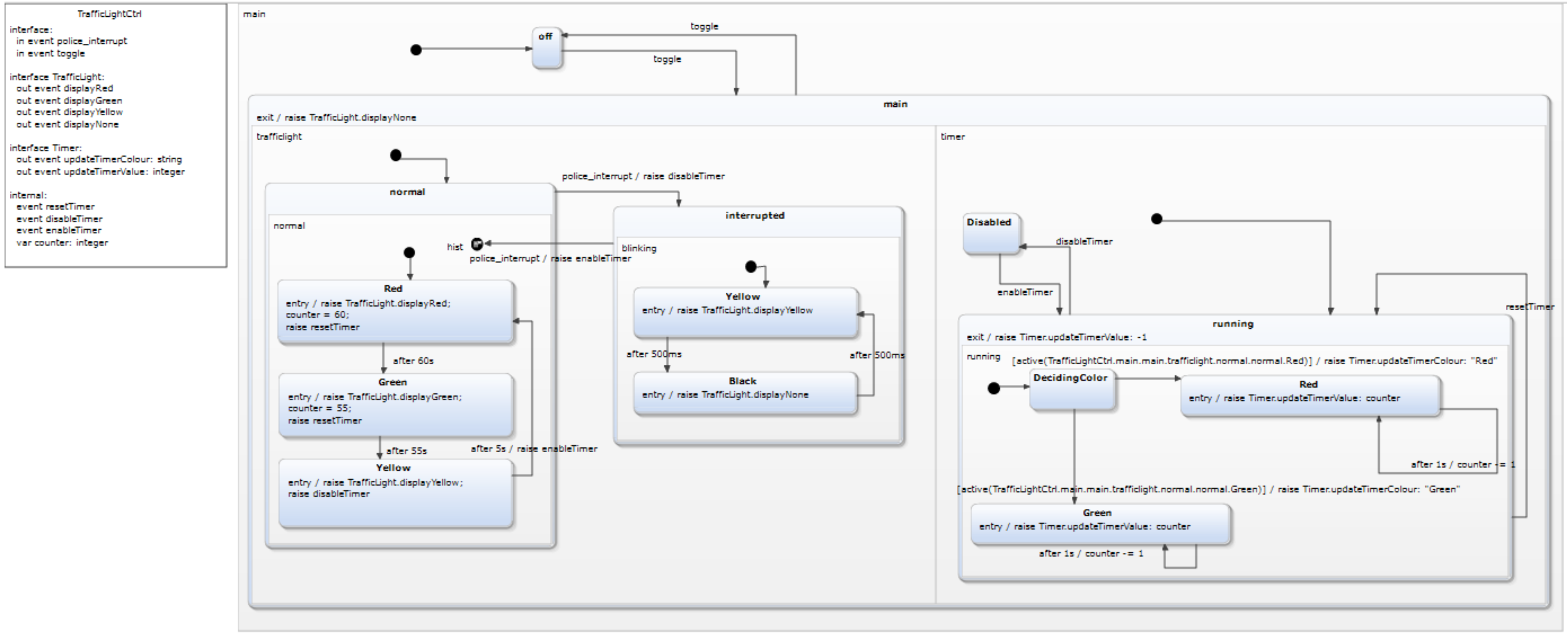

:GatherReqs
M:Statecharts
:Text. Req.
:ModelSystem
:ReviseSystem
Verify System
:DefineInput
:DefineTestCases
IT: TraceLang
TC: Statecharts
:Simulate
:Test
ST: TraceLang
TT: TraceLang
:Boolean
:CheckOutput
[False]
:Boolean
[True]
[False]
[True]
:GenerateCode
:Java
model artefact
manual activity
automatic activity

Code Generation



interrupts →

← events



Interface:

- setRed(boolean)
- setGreen(boolean)
- setYellow(boolean)
- setTimerValue(int)
- setTimerColour(string)

Interface:

- in event *police_interrupt*
- in event *toggle*
- out event *updateTimerColour:* string
- out event *updateTimerValue:* int
- out event *displayRed*, *displayYellow*, *displayGreen*, *displayNone*

Sample

Generated Code

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

```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();
}
```

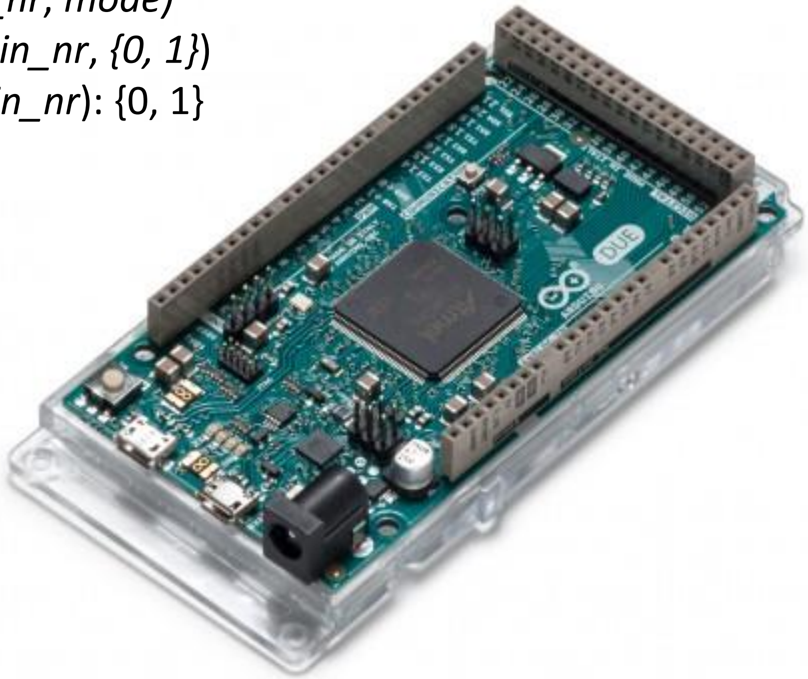
Runner

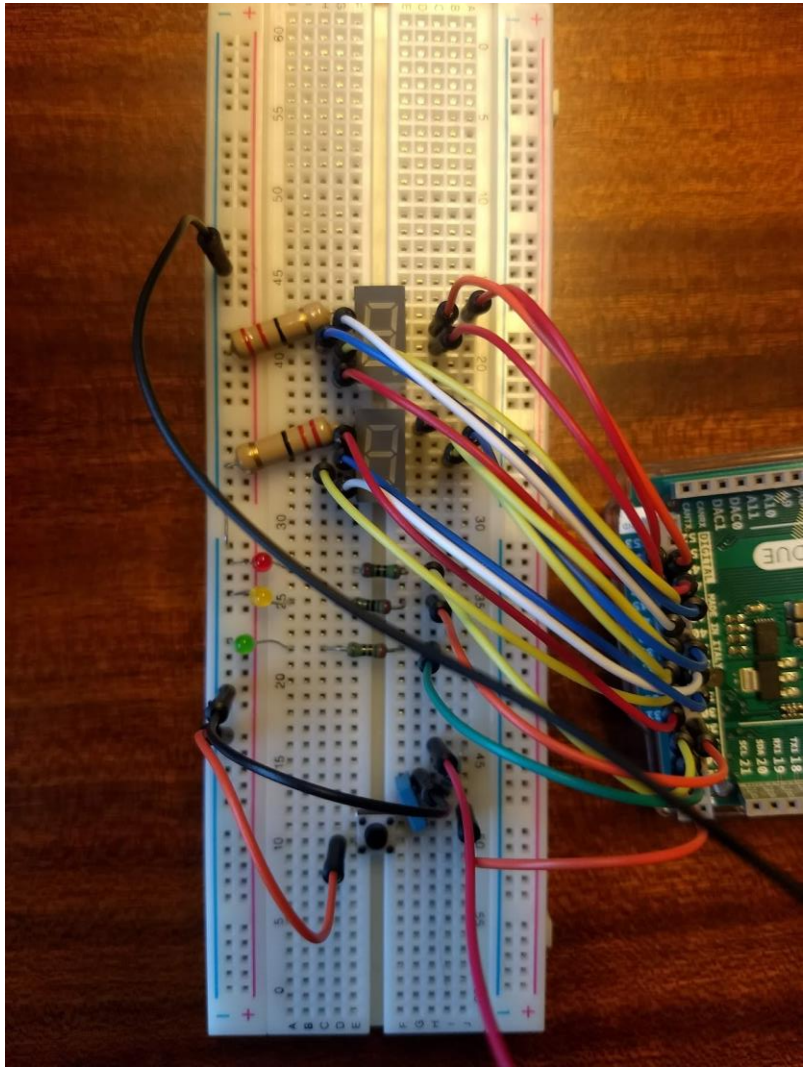
```
protected void run() {
    statemachine.enter();
    RuntimeService.getInstance().registerStatemachine(statemachine, 100);
}
```

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

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
        }

    }
}
```

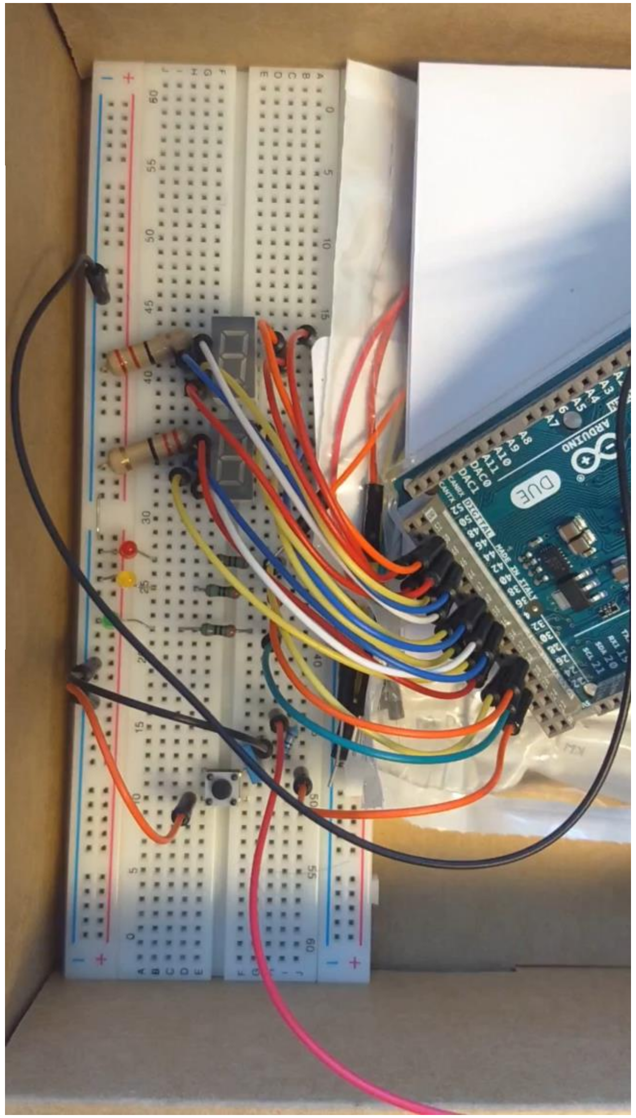
Runner

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

void loop() {
  unsigned long current_millies = millis();
  read_pushbutton(&pushbutton);
  if ( cycle_count == 0L || (current_millies >= last_cycle_time + CYCLE_PERIOD) ) {
    sc_timer_service_proceed(&timer_service, current_millies - last_cycle_time);
    synchronize(&trafficLight);
    trafficLightCtrl_runCycle(&trafficLight);
    last_cycle_time = current_millies;
    cycle_count++;
  }
}
```

Button Code

```
void read_pushbutton(pushbutton_t *button){
  int pin_value = digitalRead(button->pin);
  if (pin_value != button->debounce_state) {
    button->last_debounce_time = millis();
  }
  if ((millis() - button->last_debounce_time) > button->debounce_delay) {
    if (pin_value != button->state) {
      button->state = pin_value;
      button->callback(button);
    }
  }
  button->debounce_state = pin_value;
}
```

Deployed Application



INTRODUCTION

STATECHARTS BASICS

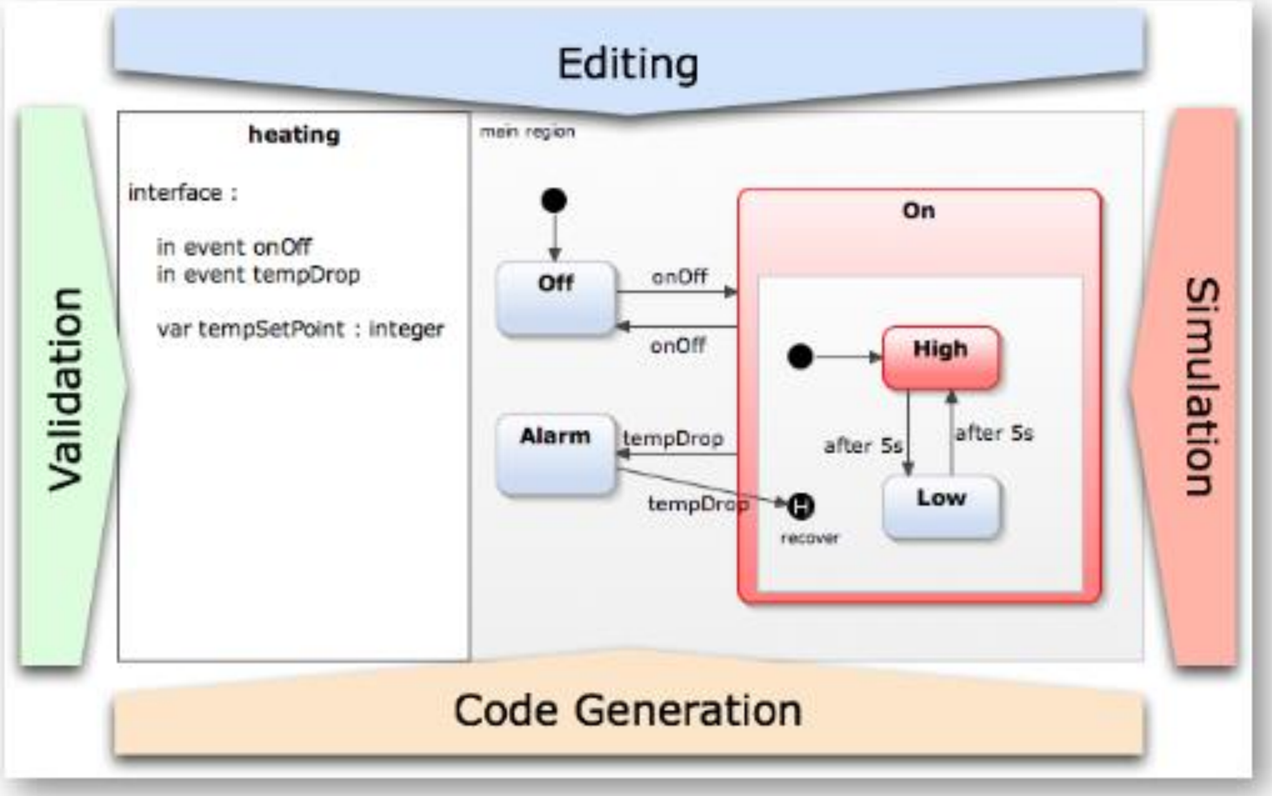
YAKINDU IN DEPTH

ADVANCED CONCEPTS

Overview

graphically create and edit Statecharts

validate syntax and (static) semantics

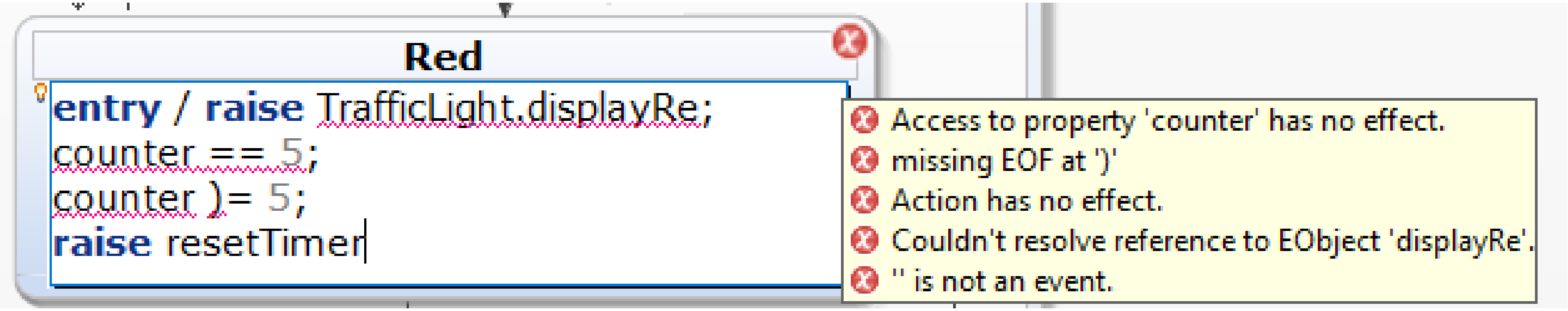


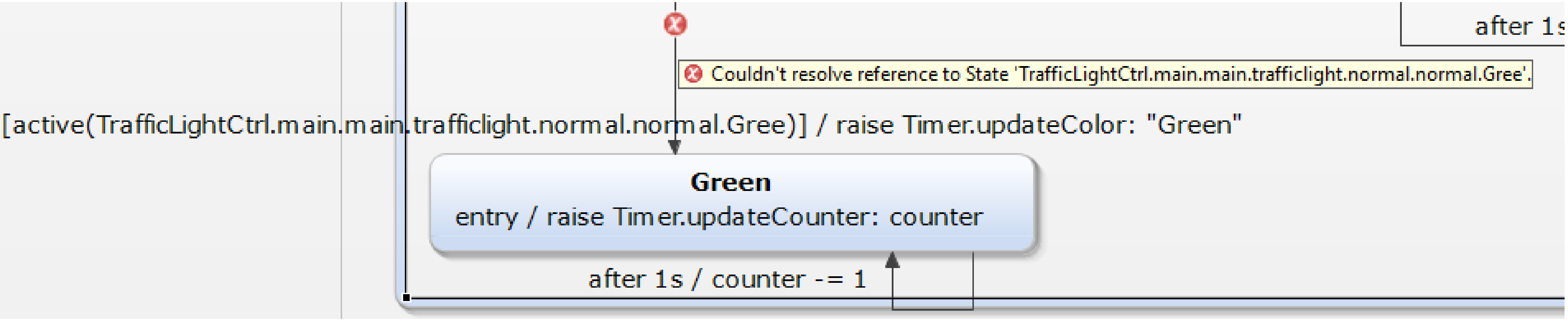
simulate the behavior of Statecharts

code generators for Java, C, and C++
+ custom code generators

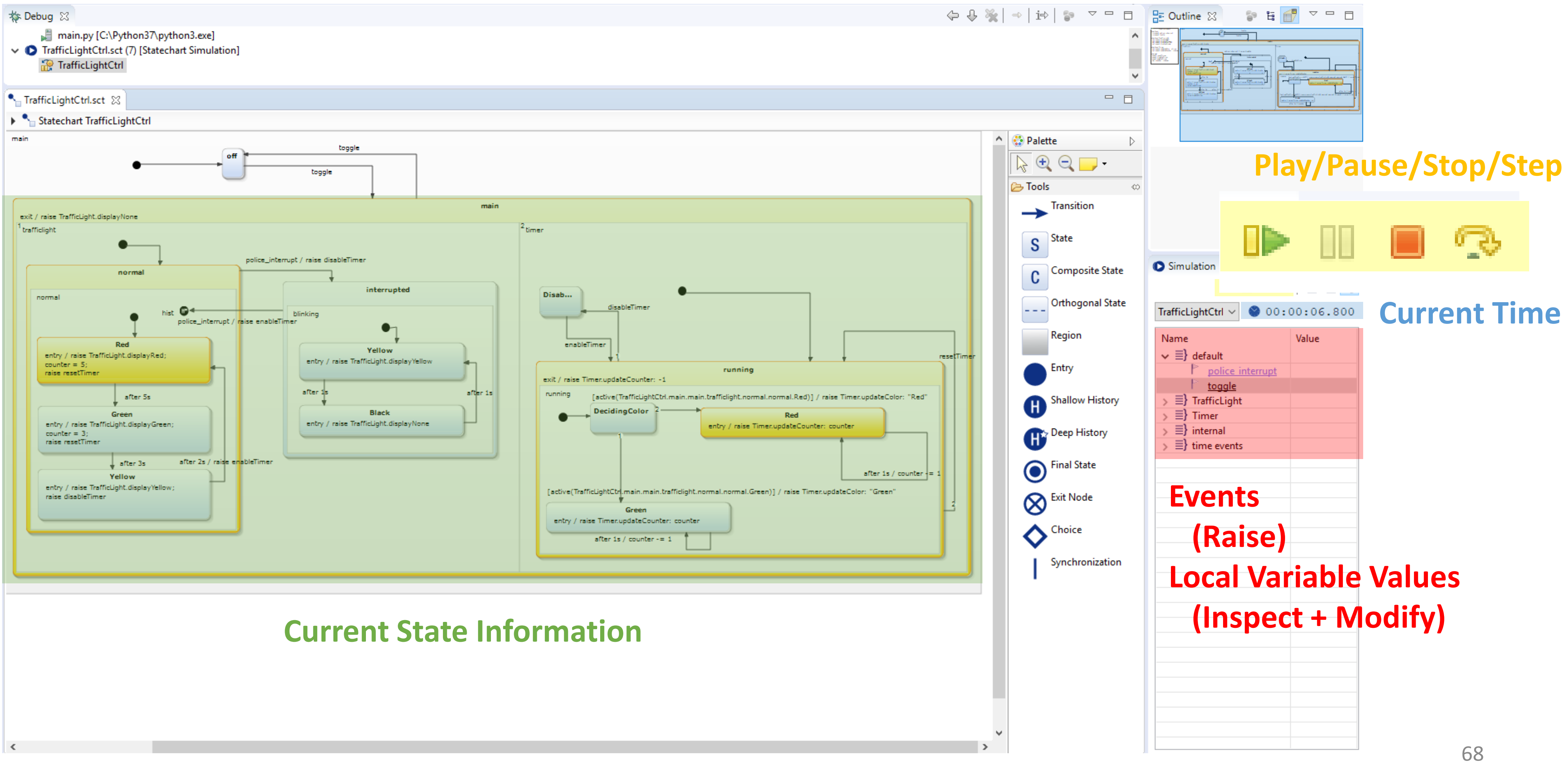
Validator

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

interface Timer:
  out event updateColor: string
  out event updateCounter: integer

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

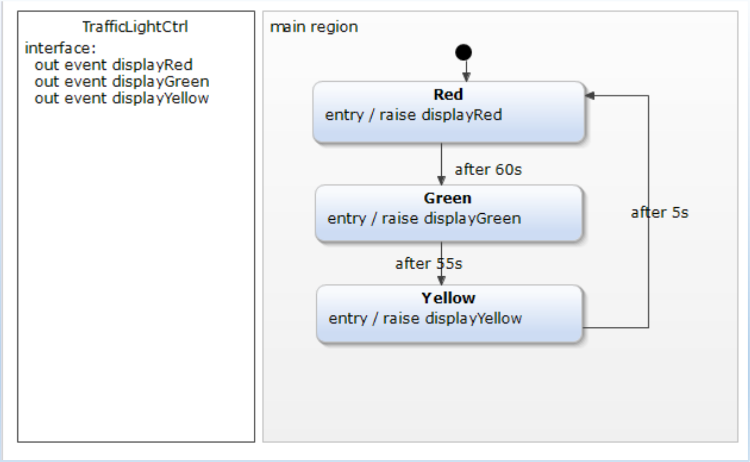




Simulation and Debugging



Code Generators



Model
(.sct file)

Generator:
- C/C++
- Java
- Custom

Configured
Code Generator

Code

Configuration
(.sgen file)

default.sgen

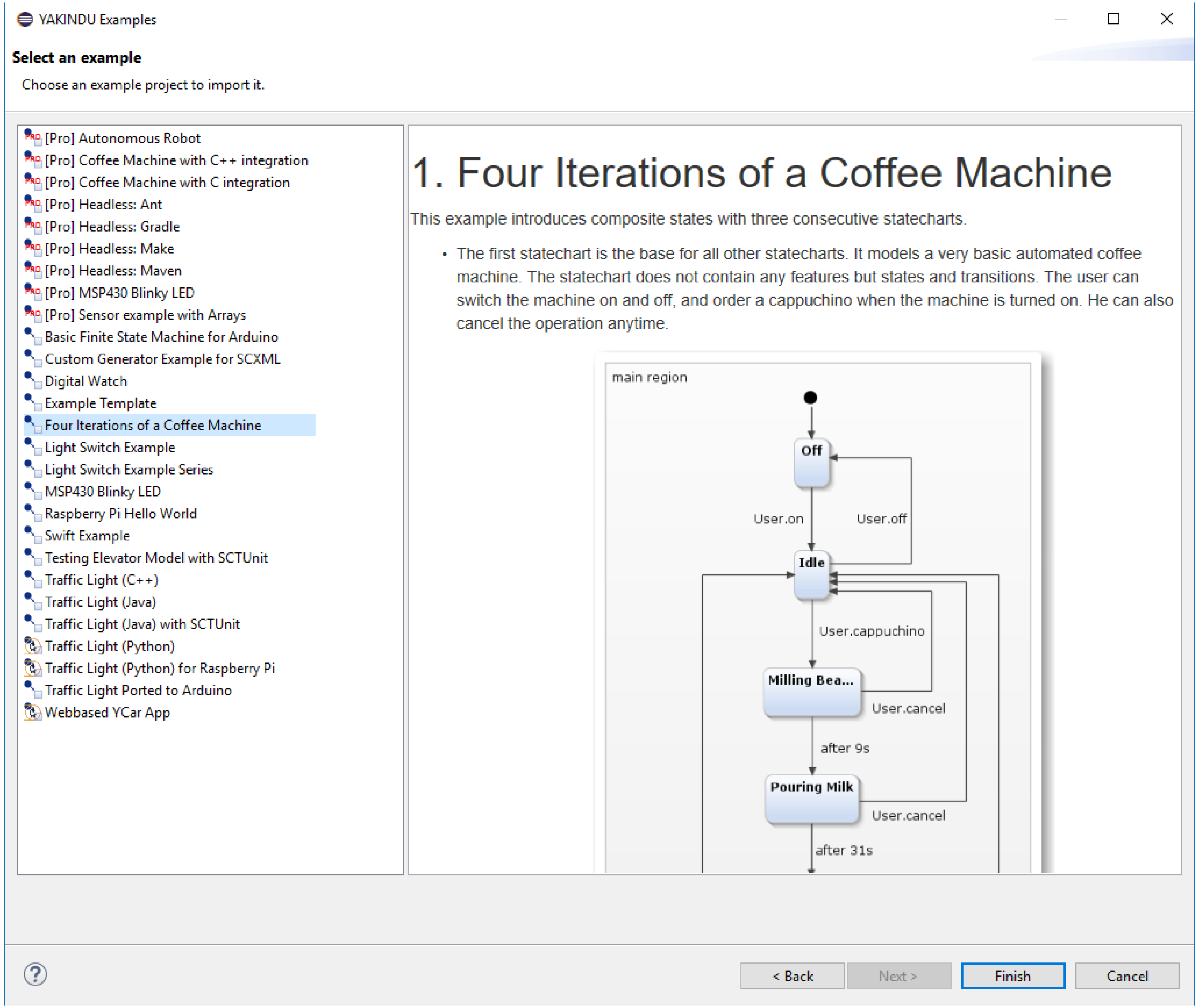
```
GeneratorModel for yakindu::java {

    statechart Statechart2 {

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

        feature JavaTargetFeatures {
            basePackage =  "org.yakindu.sct.Statechart2"
            implementationSuffix =  "Impl"
        }
    }
}
```

```
GeneratorModel for [GeneratorId] {
    statechart [StatechartReference] {
        feature [Feature] {
            [ParameterName] = [ParameterValue]
        }
    }
}
```

Examples

File -> New... -> Example... ->

Yakindu Statechart Examples



Neutral Action Language

- Types: integer/real/boolean/string/void
- Statements:
    - Assignment
    - Event Raising
    - Operation Call
- Expressions: arithmetic/condition/logical
- Built-in Functions:
    - *valueOf*(event): <value>
    - <var> *as* <type>
    - *active*(state): boolean

Neutral Action Language

trigger [*guard*] / *effect*

- Trigger:
  - after: execute after a given time (s, ms, us, ns)
  - every: execute periodically after a given time (s, ms, us, ns)
  - always: execute always
  - oncycle: same as always
  - else: useful for choice states
  - default: same as else
  - entry: execute upon entering the state
  - exit: execute upon exiting the state
- Guard:
  - Expression (boolean!)
- Effect:
  - Statement
  - Event Raise

Operation Callbacks

Model (Excerpt)

```
TrafficLightCtrl

interface TrafficLight:
  var red:boolean
  var yellow:boolean
  var green:boolean

interface Pedestrian:
  var request:boolean
  var red:boolean

  var green:boolean

interface:
  in event pedestrianRequest

  in event onOff
  operation synchronize() : void

internal:
  every 200ms / synchronize
  event blinkOff
  event blinkOn
```

Generated Code (Excerpt)

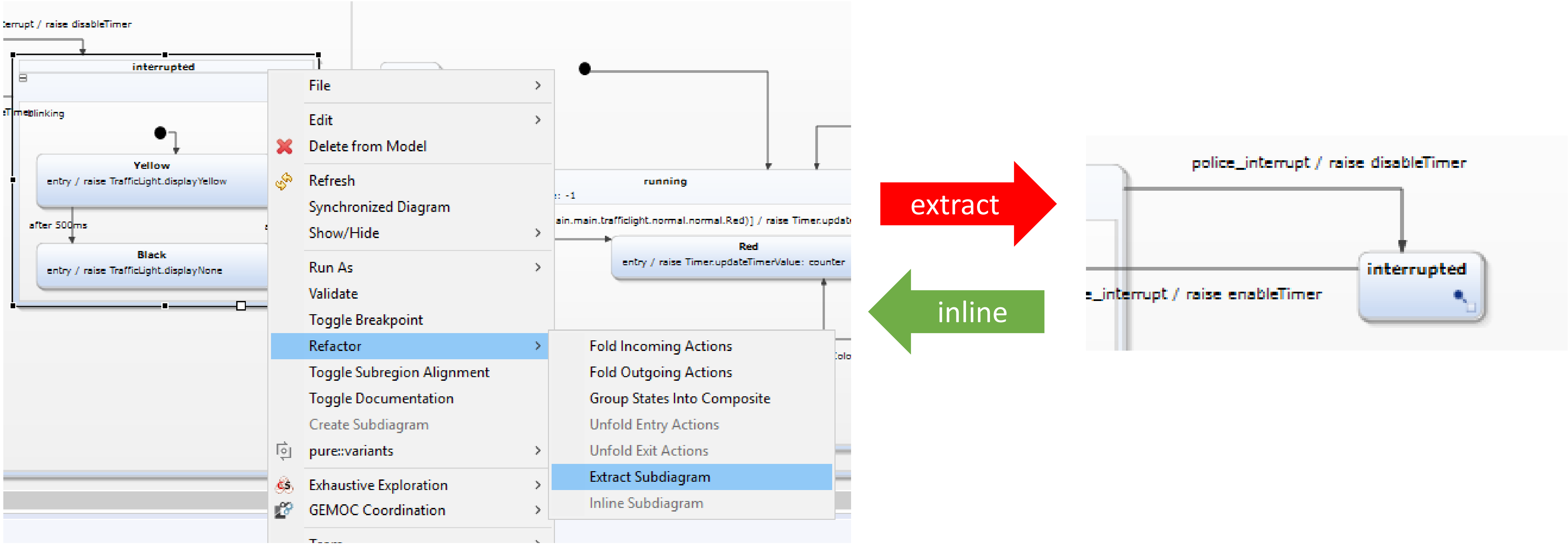
```
public interface SCInterface {

    public void raisePedestrianRequest();

    public void raiseOnOff();

    public void setSCInterfaceOperationCallback(SCInterfaceOperationCallback operationCallback);

}

public interface SCInterfaceOperationCallback {

    public void synchronize();

}
```

Runner (Excerpt)

```
statemachine.getSCInterface().setSCInterfaceOperationCallback(
        new ITrafficLightCtrlStatemachine.SCInterfaceOperationCallback() {
            @Override
            public void synchronize() {
                checkTrafficLightStates();
                repaint();
            }
});
```

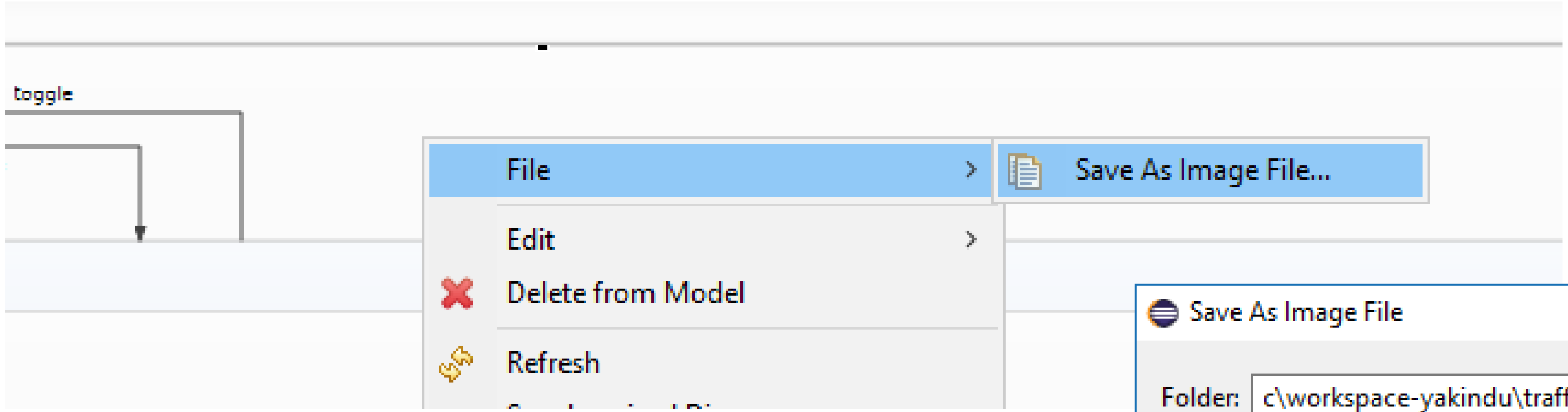
Editor Tricks

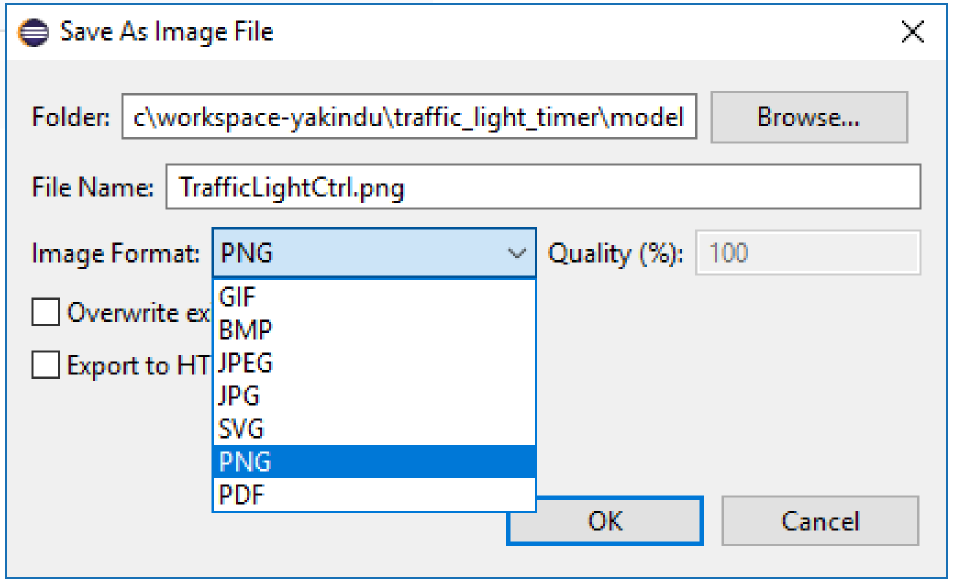
Subdiagrams



Editor Tricks

Export Model as Image





INTRODUCTION

STATECHARTS BASICS

YAKINDU IN DEPTH

ADVANCED CONCEPTS

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