Proceedings of the 2019 Winter Simulation Conference
N. Mustafee, K.-H.G. Bae, S. Lazarova-Molnar, M. Rabe, C. Szabo, P. Haas, and Y.-J. Son, eds.

INTRODUCTION TO STATECHARTS MODELING, SIMULATION, TESTING, AND DEPLOYMENT

Simon Van Mierlo
Hans Vangheluwe

Department of Mathematics and Computer Science
University of Antwerp - Flanders Make vzw
Middelheimlaan 1
Antwerp, 2020, BELGIUM

ABSTRACT

Statecharts, introduced by David Harel in 1987, is a formalism used to specify the behavior of timed, autonomous, and reactive systems using a discrete-event abstraction. It extends Timed Finite State Automata with depth, orthogonality, broadcast communication, and history. Its visual representation is based on higraphs, which combine hypergraphs and Euler diagrams. Many tools offer visual editing, simulation, and code synthesis support for Statecharts. Examples include STATEMATE, Rhapsody, Yakindu, and Stateflow, each implementing different variants of Harels original semantics. This tutorial introduces modeling, simulation, testing, and deployment of Statecharts. We start from the basic concepts of states and transitions and explain the more advanced concepts of Statecharts by extending a running example (a traffic light) incrementally. We use Yakindu to model the example system. This is an updated version of the paper with the same name that appeared at the Winter Simulation Conference in 2018 (Van Mierlo and Vangheluwe 2018).

1 INTRODUCTION

The systems that we analyze, design, and build today are characterized by an ever-increasing complexity. This complexity stems from a variety of sources, such as the complex interplay of physical components (sensors and actuators) with software, the large amounts of data these systems have to process, the non-deterministic environment with which they have to interact, etc. Almost always, however, complex systems exhibit *event-driven* behavior: the system *reacts* to stimuli coming from the environment (in the form of *input events*) by changing its internal *state* and can influence the environment through *output events*. Such event-driven systems are fundamentally different from more traditional software systems, which are *transformational* (i.e., they receive a number of input parameters, perform computations, and return the result as output). Reactive systems run continuously, often have multiple concurrently executing components, and are reactive with respect to the environment. An example is a modern car, whose systems are increasingly controlled by software. Multiple concurrently running software components interpret signals coming from the environment (the driver's controls as well as sensors interpreting current driving conditions) and making (autonomous) decisions that generate signals to the car's actuators.

Such complex event-driven behavior needs to be specified in an appropriate language, in order to validate the behavior with respect to its specification (using verification and validation techniques, such as testing, formal verification, and model checking), and to ultimately deploy the software onto the system's hardware components. Traditional programming languages were designed with transformational systems in mind, and are not well-suited for describing timed, autonomous, reactive, concurrent behavior. In fact, describing complex systems using infrastructures provided by the operating system such as threads and semaphores quickly results in unreadable, incomprehensible, and unverifiable program code (Lee 2006).

This is partly due to the cognitive gap between the abstractions offered by the languages and the complexity of the specification, as well as the often ill-defined semantics of programming languages, which hampers understandability. As an alternative, this tutorial describes the Statechart formalism, introduced by David Harel in 1987 (Harel 1987). A formalism is a language offering abstractions whose semantics are well-defined; analysis of the models is possible without having to resort to compiling and executing them to observe their behavior. A formalism's syntax, often in the form of a metamodel, can be instantiated by a modeler to create an instance (model) of the formalism. Such models can be executed by appropriate interpreters, compiled, analyzed, ... that were constructed based on the definition of the formalism semantics. The syntax and semantics of Statecharts are well-defined and can natively describe a system's timed, autonomous, and reactive behavior. Its basic building blocks are states and transitions between those states. States can be combined hierarchically into composite states, or orthogonally into concurrent regions. Many (visual) modeling tools exist that support the complete life-cycle of modeling a system's behavior using Statecharts: from design to verification and validation, and ultimately deployment (code generation).

Throughout the sections, we introduce the constructs of the Statechart formalism by incrementally building the model of the behavior of an example system: a traffic light. We explain the syntax as well as the semantics of each construct. The examples are modeled in the Yakindu (https://www.itemis.com/en/yakindu/state-machine/) tool, but the techniques can be transferred to any Statecharts modeling, simulation, and testing tool with comparable functionality.

Section 2 explains how a system's behavior can be observed and described using a discrete-event abstraction. Section 3 explains the running example of the tutorial: a traffic light that autonomously changes its light according to a fixed schedule, but can be interrupted by a policeman if the traffic is to be controlled manually. Section 4 introduces the basic building blocks of a Statechart model: states and transitions. Section 5 explains how states can be combined into composite states and in orthogonal regions, as well as history states and a number of constructs in the Statechart formalism that make the modeler's life easier, but can be considered "syntactic sugar". Section 7 explains how code can be generated from a Statechart model, and how it can be deployed onto multiple platforms. Section 8 concludes the tutorial.

2 DISCRETE-EVENT ABSTRACTION

Certain system behavior, in particular the behavior of control software, can be described using a discrete-event abstraction. In Figure 1 a view of the behavior of an example system, a "tank wars" game is shown – a tank drives around a virtual map by reacting to a player's input through the keyboard. The tank can shoot at the player's command, and it can run out of fuel, at which point it goes into a mode where it can only drive towards a fueling station (and is no longer able to shoot).

As is clear from this intuitive description, the system reacts to *input* from the environment, and produces *output* to the environment. Such input/output signals can be described by *events*. At the top of Figure 1, an input event *segment* is shown. A segment is a series of events over a finite interval of time. Within such a finite interval, only a finite number of such events can occur (which differentiates discrete-event systems from continuous systems, whose input and output behavior we can infinitely zoom into, as they are continuous functions). The system *reacts* to the input event segment by producing an output event segment, shown at the bottom of Figure 1. The environment (entities interacting with the system) can view the system as a black-box which has an interface (defined by the input events it accepts, as well as the output events it produces). In this case, the player interacts with the system by sending input events corresponding to key strokes: the player controls the tank by pressing the *up*, *down*, and *enter* key. As a result, the system produces output, which describes the reaction of the system to the input it receives. In this case, four output events can be produced: *move_up*, *move_down*, and *shoot*, which signify that the tank starts moving up, starts moving down, or shoots, respectively, and *low_fuel*, which signifies that the tank is low on fuel.

The system has an internal state, which changes over time as a result of input being received, as well as autonomously by the system. A possible system state trajectory for the example system is shown in the

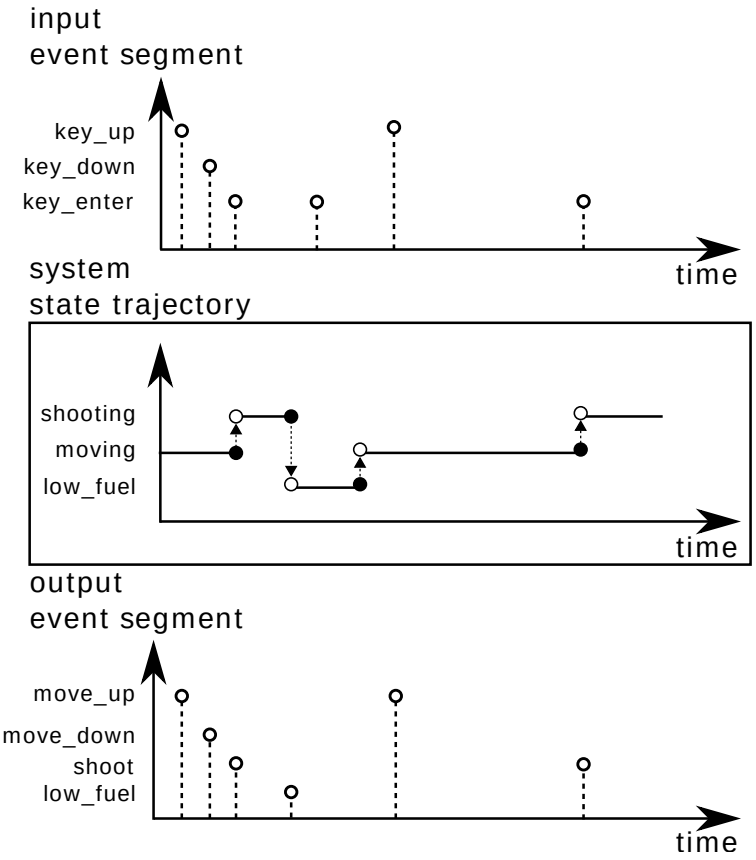


Figure 1: Discrete-event abstraction of the example "tank wars" game.

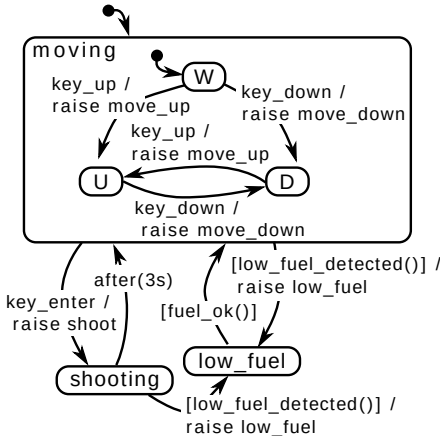


Figure 2: A possible behavioral model of the example "tank wars" game.

middle of Figure 1. Three states are defined: *shooting*, *moving*, and *low_fuel*. The first two input events do not cause the state of the system to change, but it does cause two corresponding output events to be raised by the system. The third input event changes the system state from *moving* to *shooting*. And, at some point after that, the system autonomously changes its state to *low_fuel*. From the input, output, and state trace, we can deduce that the system is *timed*, *reactive*, and *autonomous*.

To describe all possible state trajectories of the system, a state diagram can be used – see Figure 2 for a possible model describing the system's behavior. It shows the different states or *modes* the system can be in: at the highest level, three states (*moving*, *shooting*, and *low_fuel*) are defined (represented by rounded rectangles). The *moving* state has three substates, corresponding to the direction the tank is traveling in. The state of the system can change when a transition (represented by an arrow) *triggers*. A transition triggers due to an (optional) *event* or *timeout*, and an optional *condition* on the total state of the system (including the values of the system's variables). When a transition is triggered, an action is executed, which can change the values of the system's variables, or *raise* an event.

This concludes a high-level description of discrete-event abstractions to describe a system's behavior, including a possible diagrammatic notation. In the rest of this paper, the *Statechart* formalism is explained as an example of such a diagrammatic language to describe the timed, reactive, and autonomous behavior of systems.

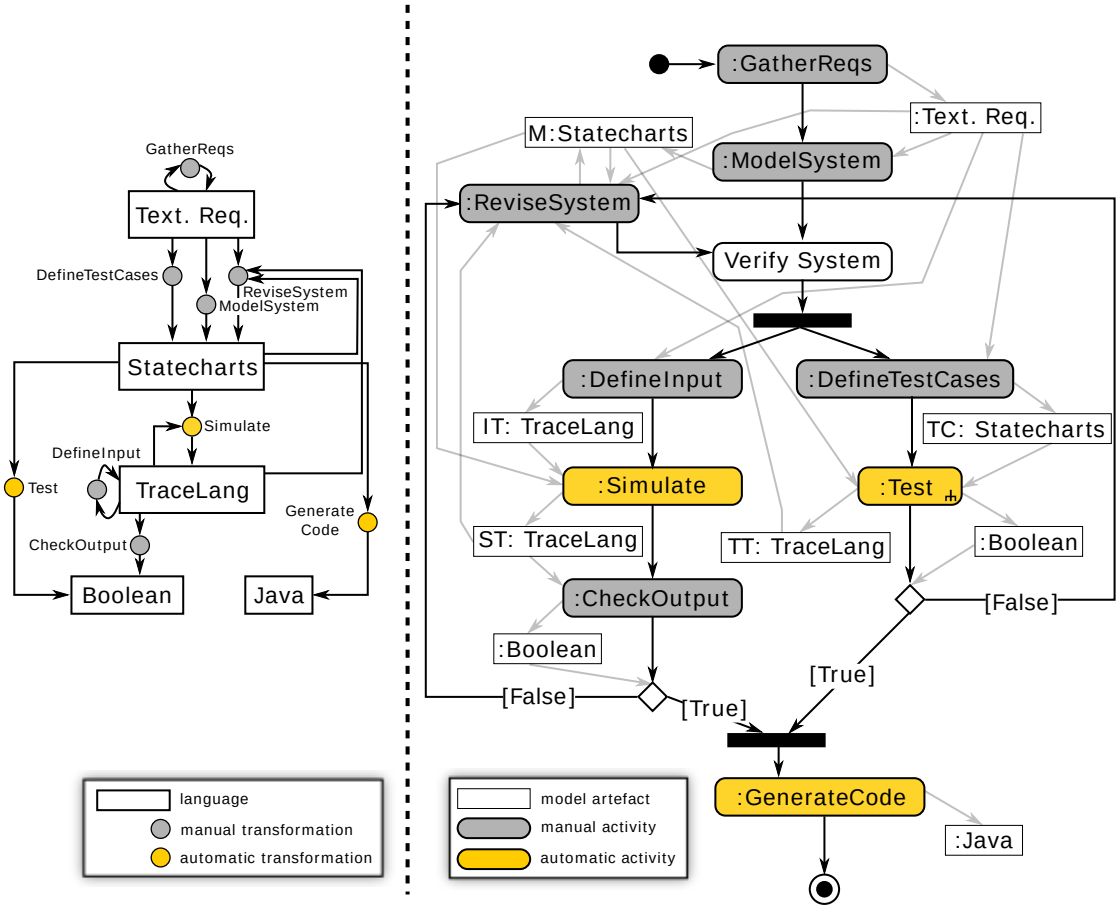


Figure 3: The workflow for modeling, simulating, testing, and ultimately generating code from Statechart models.

3 RUNNING EXAMPLE

As a running example, we will develop a Statechart model describing the timed, autonomous, and reactive behavior of a traffic light. Whenever a system is developed, however, it is important to consider the artefacts created during development and to describe the *process* which governs system development. This process, or workflow, will guide us through the tutorial to design, simulate, test, and ultimately deploy our example system. Figure 3 shows a model of this process, in a *Formalism Transformation Graph and Process modeling* (FTG+PM) language (Lúcio et al. 2013).

On the right side, a process model (PM) describes the different phases in developing the system. The process consists of several *activities*, either manual or automatic. Manual activities require user input: for example, creating a model starts with a user opening a model editor and ends when the user saves the model and closes the model editor. Automatic activities are programs that are transformational, in the sense that they can be seen as black boxes that take input and produce output. All activities produce *artefacts*, and can receive artefacts as input. Fork and join nodes can split the workflow into parallel branches, where multiple activities are active at the same time. Decision nodes can decide, depending on a boolean value, how the process proceeds.

On the left side, a formalism transformation graph (FTG) represents a map of all the *formalisms* used during system development. Each artefact produced in the process model conforms to a formalism's syntax in the formalism transformation graph. Moreover, it defines *transformations* between the formalisms, which can either be manual or automatic. Again, there is a correspondence between activities in the process

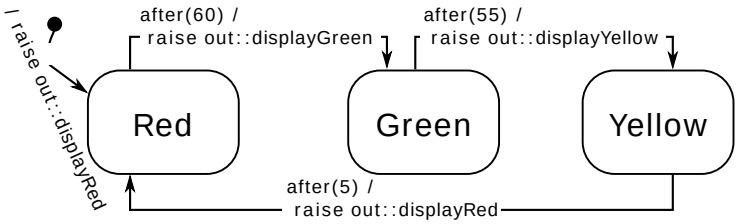


Figure 4: The basic model of the traffic light's behavior.

model and transformations in the formalism transformation graph: the transformations act as an "interface" defining the input and output artefacts, to which the activities in the workflow need to conform.

In our workflow, we will start by defining the requirements of the example system and developing an initial model of the system. This model is subsequently (and in parallel) simulated and tested. A simulation produces an output trace from a given input trace (see the previous section for a discussion on discrete-event abstraction). This output trace is manually checked and a decision is made whether or not the tested requirements are satisfied. In the other parallel branch of the workflow, a test case is defined by a generator (which produces input events) and an acceptor, which checks whether the generated output trace is correct. A test runs fully automatically, and again a decision is made whether the tested requirements are satisfied by the design of the system. If that is not the case, the model of the system is revised until all requirements are satisfied. Once all requirements are satisfied, the system can be deployed by generating appropriate application code.

To complete the first step in the workflow, the requirements for the traffic light are listed below:

1. There are three differently colored lights: red, green, and yellow.
2. At most one light is on at any point in time.
3. At system start-up, the red light is on.
4. The traffic light cycles through red on, green on, and yellow on.
5. The red light is on for 60s, the green light is on for 55s, and the yellow light is on for 5s.
6. The police can interrupt the traffic light's autonomous operation. This results in a blinking yellow light (on for 1 second, and off for 1 second repeatedly).
7. The police can resume an interrupted traffic light. The result is that the light which was on at time of interrupt is turned on again.
8. A timer displays the remaining time while the light is red or green; this timer decreases and displays its value every second. The color of the timer reflects the color of the traffic light.

In the next sections, the model of the system will be incrementally developed, which introduces both the syntax and the semantics of the different elements in the Statechart formalism.

4 BASIC BUILDING BLOCKS

The basic building blocks of any Statechart model are *states* and *transitions* between those states. They are essential concepts that need to be explained before moving onto more advanced Statechart elements. These basic building blocks have a theoretical underpinning in Finite State Automata (Hopcroft et al. 2006). To illustrate the use of states and transitions, a basic model of a traffic light (implementing the first five requirements listed in the previous section) is presented in Figure 4.

4.1 States

States model the *mode* a system is in. In the absence of concurrent regions, exactly one state is active at any point in time of the system's execution. A state has a *name*, uniquely identifying it. Exactly one state is the initial state – on system start-up, the state of the system is initialized to that initial state. The visual

representation of a state is a rounded rectangle, or roundtangle. To visualize the initial state, a small black circle is drawn, with an arrow pointing to the initial state.

In Figure 4, three states of the traffic light are modeled, corresponding to the possible colors of the traffic light: *Red*, *Green*, and *Yellow*. States in themselves have no semantics; no link to the actual color of the physical traffic light is made yet. However, aptly naming states is important, and, therefore, we can mentally make the connection that if the system is in the *Red* state, the red light will be on, and the others will be off, and similarly for the *Green* and *Yellow* states.

Besides the state – or mode – the system is in, a system keeps track of a number of *state variables* $\{v_1, v_2, \ldots, v_i\}$. The data type and possible assignments for these variables depend on the data model supported by the specific variant of the Statechart formalism. In case of Yakindu, the implementation-language-independent types *integer*, *real*, *boolean*, *string*, and *void* are defined (https://www.itemis.com/en/yakindu/state-machine/documentation/user-guide/sclang_statechart_language_reference#sclang_types).

4.2 Transitions

While states describe the current configuration the system is in, transitions model the dynamics of the system and describe how this configuration evolves over time. A transition connects exactly two states: the *source* state and the *target* state. When the system is running, a transition can *trigger* when its *triggering condition* is satisfied. When the transition triggers, the current state of the system is changed from the source state to the target state. At the same time, the transition's *action* is executed. In general, the signature of a transition is written as follows: $< trigger - event > [< trigger - condition >] \,/\, < action >$. The triggering condition of a transition consists of the following elements:

- A triggering *event* (optional), identified by a *name* and a list of *parameters*. In general, an event has the following signature: $< event - name > (< event - params >)$. The event can be an input event (coming from the environment), in which case the event name is preceded by *in* ::. Or, it can be internal to the Statechart model, in which case there is no prefix. The triggering event can also be a timeout, which is identified by the reserved event name *after* and a parameter denoting the amount of time (specified in a specific time base, such as seconds or milliseconds) that will pass until the timeout triggers.
- A triggering *condition* (optional), which models a boolean condition on the state of the system. For example, it can check whether the values of system variables have a specific value.

A transition that has no triggering condition is said to be *spontaneous*; its only triggering condition is its source state being active. The transition leaving the marker for the initial state is always spontaneous to ensure proper initialization of the Statechart model. Upon execution of the transition, the transition's action can:

- Raise events, either locally to the Statechart model, or as output to the environment. In general, an event has the following signature: $< event - name > (< event - params >)$. In the models presented in this tutorial, the names of events raised to the environment are preceded by *out* ::.
- Perform computations and assignments on the system's variables.

In Figure 4, transitions are modeled that describe the traffic light's timed behavior: the state of the system changes after certain delays, and an output event is raised when the state is changed, corresponding to the light that has to be activated. The *interface* of this model consists of the set of accepted input events $X = \varnothing$ and the set of possible output events $Y = \{displayRed, displayGreen, displayYellow\}$. When this system is placed in an environment and executed, the environment can listen to the output events and take appropriate actions (in this case, turning the correct lights on or off), and influence the behavior of the system by raising input events. This first version of the system is fully autonomous and does not accept input events (yet).

5 STATECHART EXTENSIONS

The Statechart formalism has a number of extensions that make it easier to develop complex systems. If only basic states and transitions were available, models would not scale and for realistic examples would consist of hundreds or thousands of states, hindering understandability. In the next subsections, we explore composite states (which allow for the nesting of states up to arbitrary depth), orthogonal regions (which allow for the modeling of concurrent behavior), and history states (which allow for the restoration of the state of a re-entered composite state). Last, we explain a number of constructs that do not add functionality, but make it easier to express certain behavior (so-called "syntactic sugar"). We use the full model of the traffic light's behavior, shown in Figure 5, to illustrate the different elements of the Statechart formalism.

5.1 Composite States

A composite state is a collection of substates, which themselves can be basic states or composite states. This allows modelers to nest states to arbitrary depths. The main purpose of composite states is to group behaviors that logically belong together. Transitions that are defined on the outer state can be thought of as being defined on any of the inner states as well – through a flattening procedure, it is possible to obtain an equivalent Statechart model that only consists of basic states and transitions. For example, in Figure 5, two high-level modes for the traffic light are defined in the composite states *normal* and *interrupted*. If a *police_interrupt* event is raised by the environment, the mode switches, regardless of the active substate. In case a transition has a composite state as target, the default state of that composite state is entered (transitively, to the lowest level). This means that all composite states need to have exactly one default state, as was the case for the Statechart model as well.

One important issue with composite states is that unwanted non-determinism can occur if a state has an outgoing transition that is triggered on the same event as an outgoing transition defined on one of its ancestor states. In the flattened version of the Statechart model, this non-determinism will be obvious, since a state will have two outgoing transitions that are triggered on the same event. For example, in Figure 5, if there was a transition on the *police_interrupt* event from the state *Red* to the state *Green*, the model is non-deterministic in case the *Red* state is active when a *police_interrupt* event is raised by the environment. To resolve such non-determinism (as Statecharts is a deterministic formalism), either the outer-most transition can be chosen – as is the case in STATEMATE (Harel and Naamad 1996) – or the inner-most transition can be chosen – as in Rhapsody (Harel and Kugler 2004). These different options are presented in Figure 6. In this tutorial, we assume STATEMATE semantics.

5.2 Concurrent Regions

States can be combined hierarchically in composite states (as explained in the previous subsection), or orthogonally in concurrent regions. While before, exactly one state of the Statechart model was active at the same time, when entering a state that has concurrent regions, all regions execute simultaneously. This means that they can react to events concurrently, and communicate with each other. This is done by raising events in one concurrent region that are "sensed" by the other concurrent regions (broadcast communication).

In the full Statechart model of the traffic light system in Figure 5, two orthogonal regions *trafficlight* and *timer* are modeled. The behavior of the first region controls the color of the traffic light, switching it from red, to green, to yellow, and back to red, and allows a policeman to interrupt the "normal" behavior (showing a blinking yellow light) by triggering on the *police_interrupt* input event. The second region controls the timer: it counts down a timer while the green or red light is active. To implement this behavior, a variable *counter* is introduced, of type *integer*. Two methods *setTimerValue* and *getTimerValue* are used to set and get the value of this counter respectively, and *decreaseTimerValue* decreases the counter by 1. The *trafficlight* component communicates with the *timer* by sending the following events:

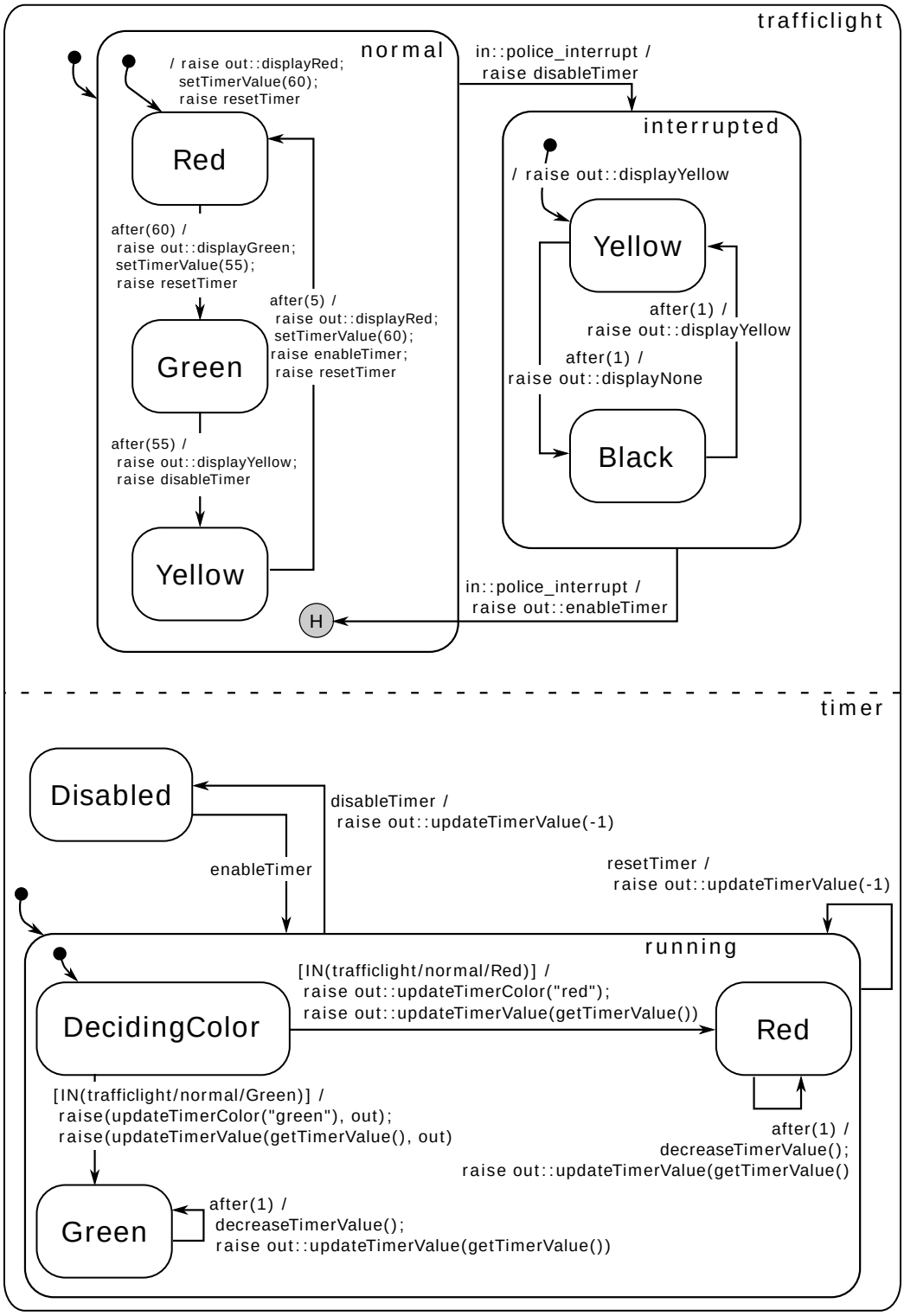
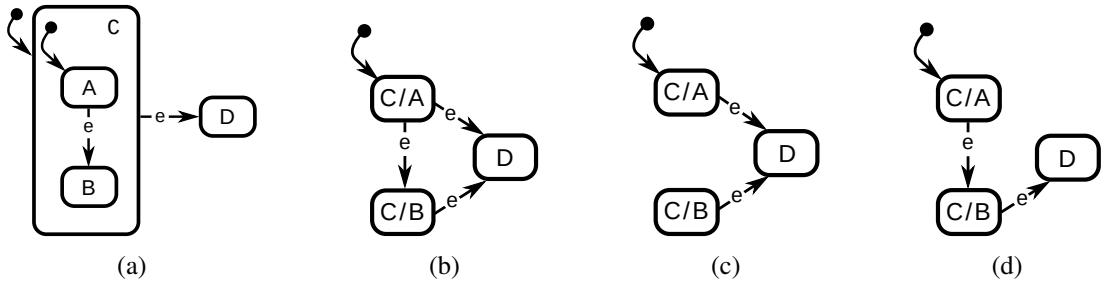


Figure 5: The full model of the traffic light's behavior.



(a) (b) (c) (d)

Figure 6: Non-determinism in composite state; (a) an example model containing non-determinism, (b) flattened version: non-determinism in state A, (c) Statemate semantics: outermost transition is prioritized, (d) Rhapsody semantics: innermost transition is prioritized.

- *resetTimer*, which signals that the timer has to be reset to the appropriate color and value.
- *disableTimer*, which signals that the the timer has to be disabled.
- *enableTimer*, which signals that the timer has to be enabled (displayed) if it's disabled.

Upon receiving one of these internal events, the timer component performs the communication with the environment by raising the following events:

- *updateTimerValue* tells the environment to update the displayed value of the timer. A negative value signifies that no value should be displayed.
- *updateTimerColour* tells the environment to update the displayed color of the timer.

5.3 History

A last element of the Statechart formalism is the *history* state. A history state can be placed in a composite state as a direct child. It remembers the current state the composite state is in when the composite state is exited. Two types of history states exist: *shallow* history states remember the current state at its own level, while *deep* history states remember the current state at its own level and all lower levels in the hierarchy. When a transition has the history state as its target, the state that was remembered is restored (instead of entering the default state of the composite state).

In Figure 5, a history state is modeled that remembers the state of the *normal* composite state when it is exited through the transition to the *interrupted* composite state. The transition that re-enters the *normal* state has the history state as its target, which restores the state that was remembered. For example, when the *Green* state was active when a *police_interrupt* state is raised by the environment, the next *police_interrupt* event will re-active the *Green* state. If the history state were not present, the *Red* state would be entered instead.

5.4 Syntactic Sugar

The previous subsections discussed the essential elements of the Statechart formalism. There are, however, additional syntactic constructs that make the modeler's life easier, but can be modeled using the "standard" Statechart constructs as well.

One of those "syntactic sugar" additions is the entry/exit action for states, which is a more efficient way of specifying actions that always need to be executed when a state is entered or exited, instead of repeating the action on each incoming/outgoing transition. An entry action is executed when a state is entered, while an exit action is executed when a state is exited. This has an important effect on the semantics of executing a transition combined with composite states. A transition is defined between states *A* and *B*. When executing this transition, the state A is exited, and the state B is entered. However, this is only the

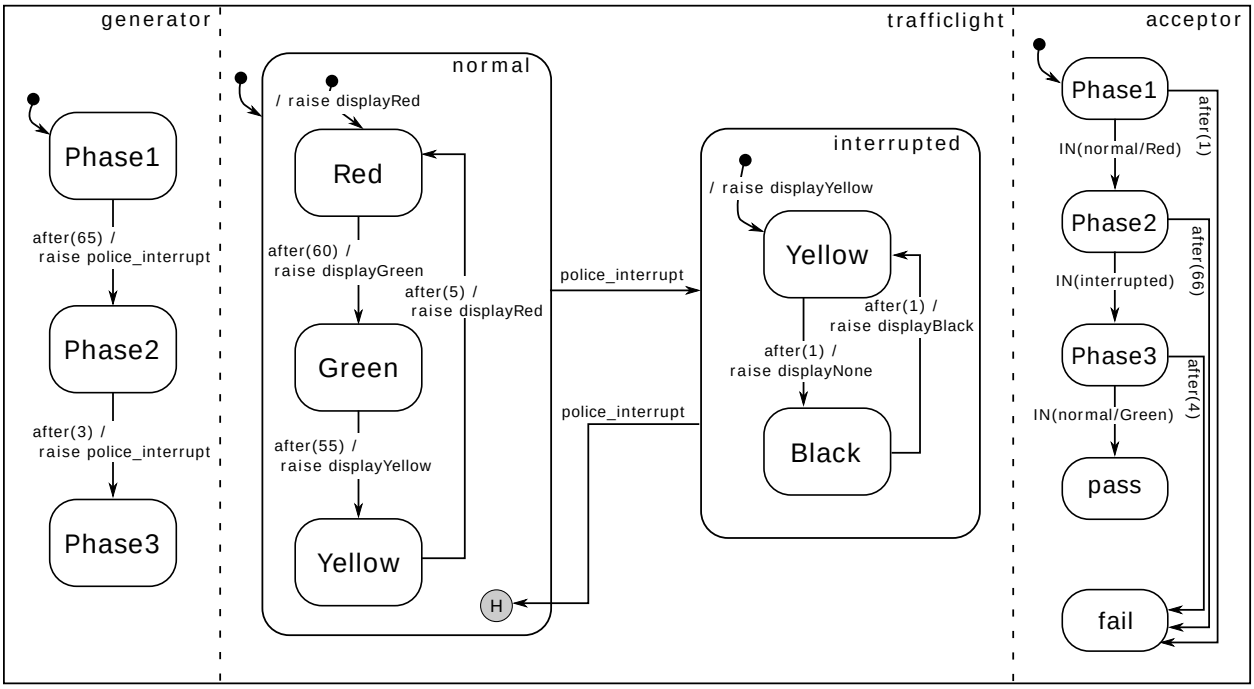


Figure 7: A test for the traffic light system (without timer).

case if A and B are part of the same composite states. More states are exited if A is part of a state hierarchy, and more states are entered in case B is part of a (different) state hierarchy. To execute a transition, the "least common ancestor" (LCA) state is computed from the source state A and target state B. The LCA is a state up the hierarchy of both A and B that has both A and B as a substate (and is the bottom-most state to have that property). To execute a transition, starting from A, the states in the hierarchy up to but not including the LCA are exited (and their exit actions are executed in the same order). Then, the transition's action is executed. Then, the states down the hierarchy towards B are entered, including B but excluding the LCA (and their enter actions are executed in the same order).

6 TESTING STATECHART MODELS

To test a Statechart model, we need to define a trace of input events and an expected trace of output events, as was shown in Figure 3. Tests need to be fully automated. Therefore, we need a different tactic from simulating the model and manually providing the input events, while checking the model's reaction. We want to autonomously generate a number of events (timed) in a "generator" and check whether the system raises the correct events in an "acceptor". Basically, an environment interacting with the system is simulated in the form of this generator-acceptor pair.

To simulate such an environment, either we regard the system as black box and use a mechanism to generate events correctly outside of the model. Alternatively, we can view the system as a white box and model the generator/acceptor pair using Statecharts as well. This has the advantage of instrumenting the model in the same language as it was developed in. Moreover, the Statechart language is appropriate to express the behavior of the generator and acceptor, as they are timed, autonomous, reactive systems. This is illustrated in Figure 7, where we develop a test case for (a part of) the traffic light model. The generator and acceptor are modeled as orthogonal regions alongside the actual system.

The test case tests the expected behavior of the traffic light when two police interrupts are raised by the environment. The generator raises a police interrupt after 65 seconds, and one more after 3 (in total 68) seconds. The acceptor checks whether the correct states are traversed by the system. First, it checks

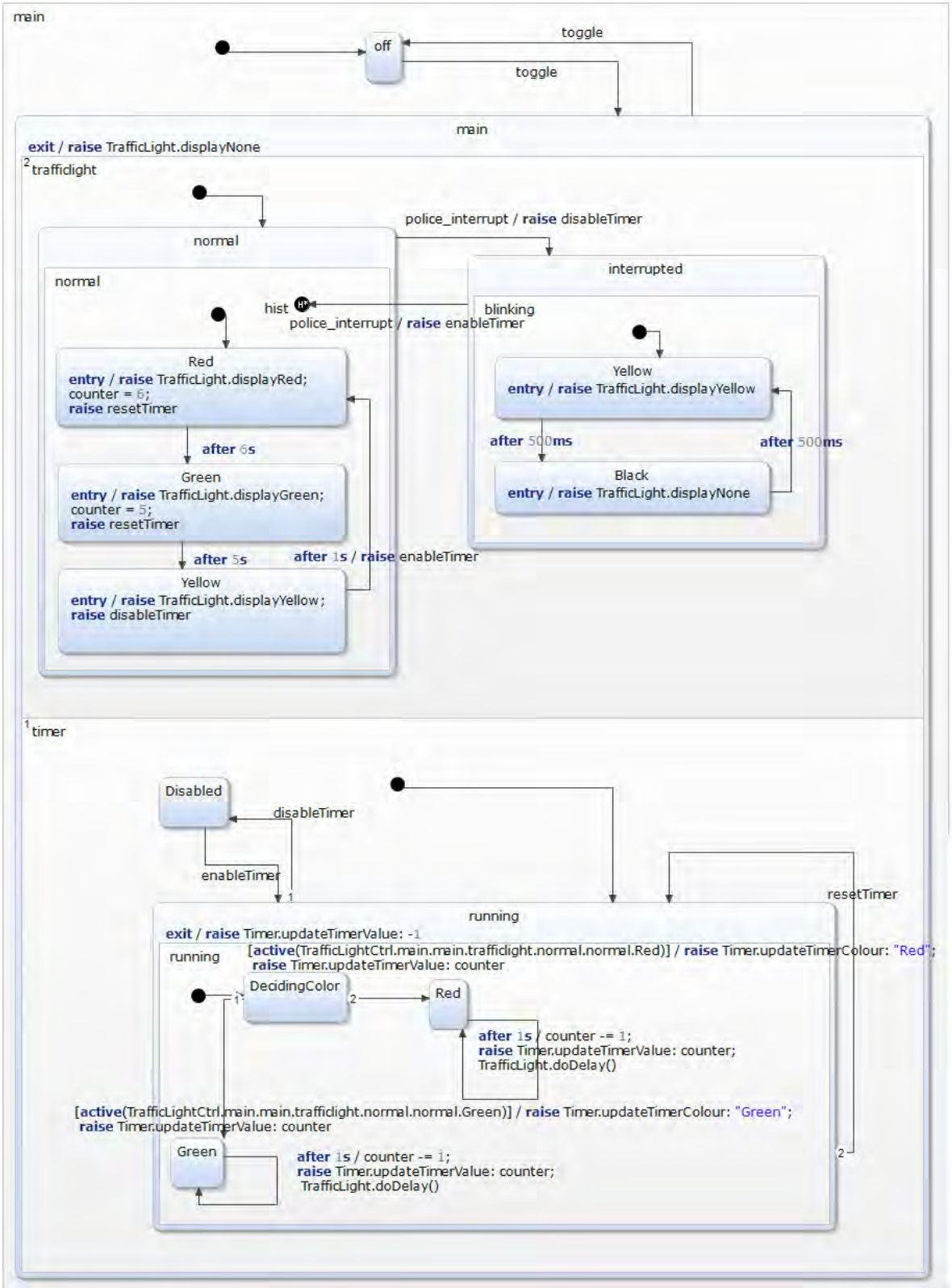


Figure 8: Screenshot of the traffic light model in the Yakindu modeling and simulation tool.

whether the system is initialized in the *normal/Red* state. Then, it checks that the *interrupted* state is entered. Last, it checks that the system's state is restored to *normal/Green* (since that was the state active when the system transitioned to the *interrupted* state). Each state has a timeout: if the system's state does not change in time (according to the time delays in its *after* transitions), the acceptor transitions to a *fail* state. The test passes if the acceptor ends up in the *pass* state.

Due to our white-box approach, we were able to both check the output events produced by the system, as well as its internal state. To be able to check the events raised by the model, we had to change these events to be locally raised, instead of raised to the environment. If we were testing using a black-box approach, the generator and acceptor could be modeled as separate Statechart models, and a communication channel between the generator, the system, and the acceptor could be set up. However, this has the disadvantage of a delay being introduced by the communication channels, which might be difficult to account for in the generator and acceptor. It does allow for testing a system for which we do not have access to the model, but it is outside of the scope of this tutorial.

7 TOOL SUPPORT: YAKINDU

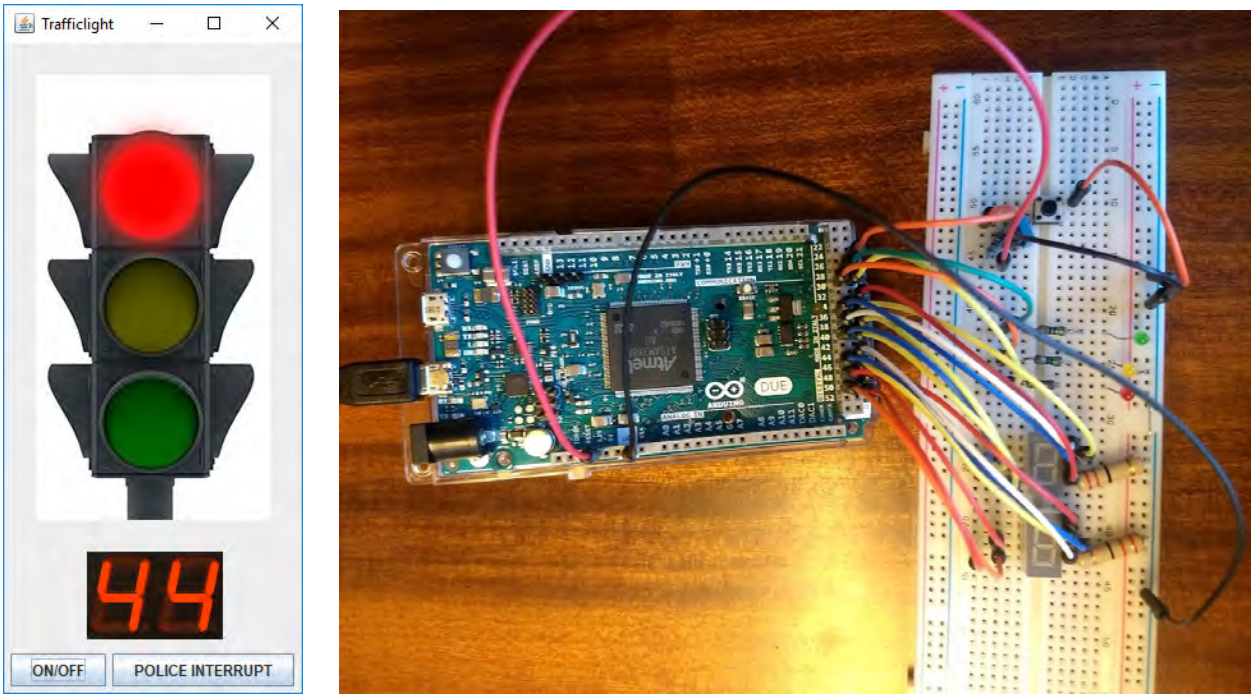
Yakindu is a Statecharts modeling and simulation tool, with the following features:

- A graphical modeling tool for describing systems with the Statechart formalism.
- A neutral action language to use in transition constraints and actions.
- A simulator, to simulate Statechart models to check its behavior. The simulator allows users to raise events while the simulation is running.
- A code generator interface for generating code to any programming language – pre-defined code generators are provided for Java, C, and C++. The code generator's configurable options include the folder to generate files in, the "execution scheme" (cycle-based or event-driven), whether listeners for external events need to be generated, etc. Yakindu allows for writing custom own code generators, increasing the flexibility of the tool.

Figure 8 shows the traffic light model of Figure 5 modeled in Yakindu. Central to the figure is the canvas, on which the Statechart model is drawn. The tool is "syntax-directed", which means only syntactically correct models can be constructed. The valid syntactic elements are shown on the right side in a palette. These elements correspond to the ones discussed in the previous sections, along with a few extra syntactic sugar elements, which will not be discussed here. On the left of the figure, an interface for the Statechart model is defined. This interface makes explicit the possible input, output, and locally raised events, as well as any data variables and operations on these variables. In the previous sections, we have left this definition of the interface implicit, but Yakindu requires to make it explicit for various reasons:

- Transition triggers can be validated, since they can only use an *after*-event or an event declared in the interface (which is either internal or external).
- Actions can be validated to only access variables that were declared, perform operations on them that are valid for their data type, and only call functions that were declared in the interface.
- When generating code, interface methods for output event listeners can be generated, corresponding to the possible output events of the system. Similarly, interface methods for raising input events (from the environment) can be generated.

By checking the syntactic validity of the model, as well as the validity of condition triggers and action code, Yakindu prevents many possible errors that a modeler can make. We can, from this model, also generate a running application. In our case, we will generate the code implementing the behavior of the traffic light for two platforms:



(a) Deploying as a virtual product (Java application).

(b) Deploying onto hardware (Arduino platform).

Figure 9: Deploying the Statechart model onto two different platforms: (a) a virtual (simulated) platform based on Java and (b) a hardware prototype based on the Arduino platform.

1. A Java-based platform, implementing the the system by providing a visualization interface that allows users to experiment with a simulated traffic light where interaction (the police interrupt) is implemented by buttons.
2. A hardware prototype based on the Arduino platform, connected to three led-lights and two seven-segment displays for displaying the state of the system, and a button for interaction.

For the Java platform, we define a visualization library that can display the state of our system. It has the following interface:

- *setRed(boolean)*: turns on the red light if the boolean is true, or off if the boolean is false.
- *setGreen(boolean)*: turns on the green light if the boolean is true, or off if the boolean is false.
- *setYellow(boolean)*: turns on the yellow light if the boolean is true, or off if the boolean is false.
- *setTimerValue(int)*: sets the value of the timer to the specified integer value. A value of -1 disables the timer.
- *setTimerColour(string)*: sets the color to the specified string, either “red” or “green”.
- *addListener(Button, Listener)*: adds a listener for the buttons in the GUI for turning on/off the traffic light, or for the police interrupt.

This library can be instantiated to show the current state of a traffic light, as is shown in Figure 9(a), where the red light is active and the 44 seconds are remaining until the light turns green. To connect this GUI to the code generated by Yakindu from the Statechart model, we define appropriate listeners for the buttons in the interface:

- The *ON/OFF* button raises a *toggle* event in the system, turning on or off the traffic light.
- The *POLICE INTERRUPT* button raises a *police_interrupt* event in the system, interrupting the normal operation of the traffic light or restoring it.

We also define appropriate listeners that translate output events raised by the system to method calls in the GUI:

- The *displayRed* event is translated to three method calls in a listener: *setRed(true)*, *setGreen(false)*, and *setYellow(false)*. Similarly, the *displayGreen*, *displayYellow*, and *displayNone* events are translated to appropriate method calls.
- The *updateTimerValue* event is translated to a method call to *setTimerValue*, passing on the correct value of the counter.
- The *updateTimerColour* event is translated to a method call to *setTimerColour*, passing on the correct color of the counter.

For the hardware prototype, we take a similar approach. The Statechart is deployed onto an Arduino platform connected to a breadboard with the required hardware, as shown in Figure 9(b). The programming language (C) is different, but a code generator is included with the Yakindu tool for generating the behavior of the traffic light. The interface methods for displaying the state of the traffic light can be programmed for the Arduino tool as well: their implementation sends a signal to the correct port (connected to the light that has to be turned on). For the interactivity, a function can be coded that reacts to the user physically pressing the button, which then raises an input event to the Statechart. Another difference with the Java platform, is that the Arduino platform schedules its tasks in a way that cannot be controlled by the user: a *loop* function within the Arduino program is called at irregular intervals. This requires the code within that function to first check whether the button was pressed, compute how much time elapsed, and to make the Statechart progress for that amount of time. Afterwards, it checks the output that was generated by the Statechart and displays the current state.

This development method allows for cleanly separating behavior (encoded in the model, and generated to executable code by an appropriate code generator) and the presentation (encoded in a visualization library). More complex control systems benefit from this by separating the control logic from the actuators and sensors, through appropriate interfaces that offer the necessary functionality.

8 CONCLUSION

In this tutorial, we introduce Statecharts as an appropriate language for describing the timed, reactive, autonomous behavior of systems. The Statechart formalism offers the following abstractions:

- States, which can be combined hierarchically into composite states or orthogonally in orthogonal regions.
- Transitions between states, encoding the dynamics of the system.
- History states, which remember the active child state(s) of a composite state when the composite state is exited.

We explained the syntax and semantics for each of the constructs and have demonstrated their use through a running example: a traffic light with a counter showing how long the traffic light's current light will stay on, and which can be interrupted by a policeman. We use Yakindu, an Eclipse-based visual modeling tool, to model the system, simulate it, test it, and ultimately generate code that is displayed on a particular platform.

ACKNOWLEDGMENTS

This research was partially supported by Flanders Make vzw, the strategic research center for the manufacturing industry.

REFERENCES

Harel, D. 1987. "Statecharts: A Visual Formalism for Complex Systems". *Science of Computer Programming* 8(3):231–274.

Harel, D., and H. Kugler. 2004. "The Rhapsody Semantics of Statecharts (or, On the Executable Core of the UML)". In *Integration of Software Specification Techniques for Applications in Engineering*, edited by H. Ehrig, W. Damm, J. Desel, M. Große-Rhode, W. Reif, E. Schnieder, and E. Westkämper, 325–354. Berlin Heidelberg: Springer.

Harel, D., and A. Naamad. 1996. "The STATEMATE Semantics of Statecharts". *ACM Transactions on Software Engineering and Methodology* 5(4):293–333.

Hopcroft, J. E., R. Motwani, and J. D. Ullman. 2006. *Introduction to Automata Theory, Languages, and Computation*. 3rd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Lee, E. A. 2006. "The Problem with Threads". *Computer* 39(5):33–42.

Lúcio, L., S. Mustafiz, J. Denil, H. Vangheluwe, and M. Jukss. 2013. "FTG+PM: An Integrated Framework for Investigating Model Transformation Chains". In *Proceedings of SDL 2013: Model-Driven Dependability Engineering: 16th International SDL Forum*, edited by F. Khendek, M. Toeroe, A. Gherbi, and R. Reed, 182–202. Berlin, Heidelberg: Springer.

Van Mierlo, S., and H. Vangheluwe. 2018. "Introduction to Statecharts Modeling, Simulation, Testing, and Deployment". In *Proceedings of the 2018 Winter Simulation Conference*, edited by M. Rabe, A. Juan, N. Mustafee, A. Skoogh, S. Jain, and B. Johansson, 306 – 320: Institute of Electrical and Electronics Engineers, Inc. Piscataway, New Jersey.

AUTHOR BIOGRAPHIES

SIMON VAN MIERLO is a post-doctoral researcher at the University of Antwerp (Belgium). He is a member of the modeling, Simulation and Design (MSDL) research lab. For his PhD thesis, he developed debugging techniques for modeling and simulation formalisms by explicitly modeling their executor's control flow using Statecharts. He is the main developer and maintainer of SCCD, a hybrid formalism that combines Statecharts with class diagrams. His e-mail address is simon.vanmierlo@uantwerpen.be.

HANS VANGHELUWE is a full professor at the University of Antwerp (Belgium). He heads the modeling, Simulation and Design (MSDL) research lab. In a variety of projects, often with industrial partners, he develops and applies the model-based theory and techniques of Multi-Paradigm modeling (MPM). His current interests are in domain-specific modeling and simulation, including the development of graphical user interfaces for multiple platforms. To model such reactive systems, he advocates the use of Statecharts to describe their behavior. His e-mail address is hans.vangheluwe@uantwerpen.be.