Chapter 6
Statecharts: A Formalism to Model, Simulate and Synthesize Reactive and Autonomous Timed Systems

Simon Van Mierlo and Hans Vangheluwe

Abstract Statecharts, introduced by David Harel in 1987, is a formalism used to specify the behaviour 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 graphs and Euler diagrams. Many tools offer visual editing, simulation, and code synthesis support for the Statechart formalism. Examples include STATEMATE, Rhapsody, Yakindu, and Stateflow, each implementing different variants of Harel's original semantics. This tutorial introduces modelling, simulation, and testing with Statecharts. As a running example, the behaviour of a digital watch, a simple yet sufficiently complex timed, autonomous, and reactive system is modelled. We start from the basic concepts of states and transitions and explain the more advanced concepts of Statecharts by extending the example incrementally. We discuss several semantic variants, such as STATEMATE and Rhapsody. We use Yakindu to model the example system.

Learning Objectives

After reading this chapter, we expect you to be able to:

- Specify the behaviour of a autonomous and reactive system using Statecharts
- Understand the semantics of a Statecharts model
- Deploy a Statecharts model onto a software platform (using appropriate tooling)

6.1 Introduction

The systems that we analyse, design, and build today are characterised 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 (real-time) software, the large amounts of data these systems have to process, and the interaction with a (non-deterministic) environment. Almost always, however, complex systems exhibit some for of *reactivity* to the environment: 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 reactive systems are fundamentally different from traditional software systems, which are *transformational* (i.e., they receive a number of input parameters, perform computations, and return the result as output). Reactive

Simon Van Mierlo
University of Antwerp - Flanders Make, Belgium
e-mail: simon.vanmierlo@uantwerpen.be

Hans Vangheluwe
University of Antwerp - Flanders Make, Belgium; McGill University, Canada
e-mail: hans.vangheluwe@uantwerpen.be

P. Carreira et al. (eds.), *Foundations of Multi-Paradigm Modelling for Cyber-Physical Systems*,
https://doi.org/10.1007/978-3-030-43946-0_6

systems run continuously, often have multiple concurrently executing components, are reactive with respect to the environment, and can act proactively by making autonomous decisions. An example is a modern car, whose systems are increasingly controlled by software. Multiple concurrently running software components are interpreting 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 timed, reactive, autonomous behaviour needs to be specified in an appropriate language, in order to validate the behaviour with respect to its specification (using verification and validation techniques, such as formal verification, model checking, as well as testing techniques), 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, and concurrent behaviour. In fact, describing complex systems using threads and semaphores quickly results in unreadable, incomprehensible, and unverifiable program code [181]. 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 sometimes ill-defined semantics of programming languages, which hampers understandability. As an alternative, this book chapter describes the Statechart formalism, introduced by David Harel in 1987 [134]. The syntax and semantics of Statecharts are well-defined and can natively describe a system's timed, autonomous, reactive, and concurrent behaviour. 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) modelling tools exist that support the complete life-cycle of modelling a system's behaviour using Statecharts: from design to verification and validation, and ultimately deployment (code generation).

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

Section 6.2 provides background for the rest of the chapter: it explains how we can view a system's behaviour using a discrete-event abstraction, it explains the behaviour of the example digital watch system we are going to model, and it introduces a process model that can be used to build a system (from requirements, through design, down to deployment) using Statecharts. Section 6.3 explains each Statechart construct, starting from the basic state and transition, and progressively introducing more advanced constructs. Section 6.4 explains the semantics of Statecharts (as implemented by STATEMATE) in detail. Section 6.5 explains how Statechart models can be tested, to check whether they satisfy the requirements of the system. Section 6.6 explains how code can be generated from a Statechart model and deployed onto a target platform. Section 6.7 goes into a number of advanced topics. Section 6.8 summarises this chapter.

6.2 Background

This chapter provides background for the rest of the chapter. In Section 8.4, we explain the running example that we will model using Statecharts in the next section. In Section 6.2.2, we look at a system's behaviour using a discrete-event abstraction. Finally, in Section 6.2.3, we present a process model that can be used to model a system using Statecharts, verify that it satisfies the requirements (through simulation and testing), and finally deploy it.

6.2.1 Running Example

The example system we use in this chapter to demonstrate the capabilities of the *Statechart* formalism is a digital watch. A visual representation of the watch is shown in Figure 6.1. A watch is primarily used for keeping time, but has other functions, such as chronometer, an alarm, and a light. A user can interact with the watch by pressing buttons that result in switching between the different modes, editing the current time, turning on the light, . . . It is inherently timed and autonomous, as it changes its internal state without any input from the user.

Fig. 6.1: The running digital watch application.

Some notion of orthogonality is also present, as the system has to ensure time is updated while the chronometer is running, for example.

A full specification of the requirements is given below:

- The time value should be updated every second, even when it is not displayed (as for example, when the chrono is running). However, time is not updated when it is being edited.
- Pressing the top right button turns on the Indiglo light. The light stays on for as long as the button remains pressed. From the moment the button is released, the light stays on for 2 more seconds, after which it is turned off.
- Pressing the top left button alternates between the time display, chrono display, and alarm display modes. The system starts in the time display mode. In this mode, the time (HH:MM:SS) and date (MM/DD/YY) are displayed.
- When in chrono display mode, the elapsed time is displayed MM:SS:FF (with FF hundredths of a second). Initially, the chrono starts at 00:00:00. The bottom right button is used to start the chrono. The running chrono updates in 1/100 second increments. Subsequently pressing the bottom right button will pause/resume the chrono. Pressing the bottom left button resets the chrono to 00:00:00. The chrono will keep running (when in running mode) or keep its value (when in paused mode), even when the watch is in a different display mode (for example, when the time is displayed).
- When in alarm display mode, the time at which the alarm is set is displayed. The default alarm time is 12:00:00.
- When in time or alarm display mode, the watch will go into editing mode when the bottom right button is held pressed for at least 1.5 seconds.
- When in time or alarm display mode, the alarm can be toggled between on or off by pressing the bottom left button.
- The alarm is activated when the alarm time is equal to the time in display mode. When it is activated, the screen will blink for 4 seconds, then the alarm turns off. Blinking means switching to/from highlighted background (Indiglo) twice per second. The alarm can be turned off before the elapsed 4 seconds by a user interrupt (i.e., if any button is pressed). After the alarm is turned off, activity continues exactly where it was left off. Note that after the alarm finishes (flashing ends), the alarm is unset (so it will not go off the next day at the same time).
- When in (either time or alarm) editing mode, briefly pressing the bottom left button will increase the current selection. Note that it is only possible to increase the current selection, there is no way to decrease or reset the current selection. If the bottom left button is held down, the current selection is incremented automatically every 0.3 seconds. Editing mode should be exited if no editing event occurs for 5 seconds. Holding the bottom right button down for 2 seconds will also exit the editing mode. Pressing the bottom right button for less than 2 seconds will move to the next selection (for example, from editing hours to editing minutes).

These requirements now need to be translated to a design of the system. Many options exist: since we are talking about control *software*, a possible choice would be a high-level programming language. But as we have argued

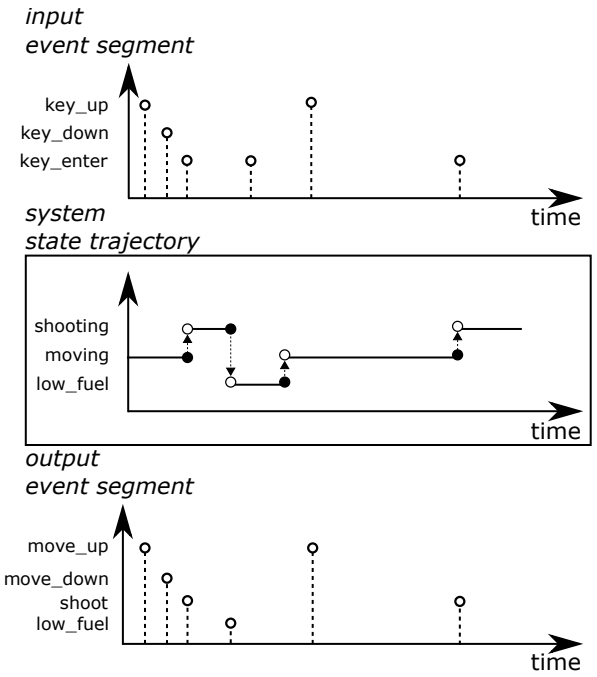

Fig. 6.2: Discrete-event abstraction of an example "tank wars" game.

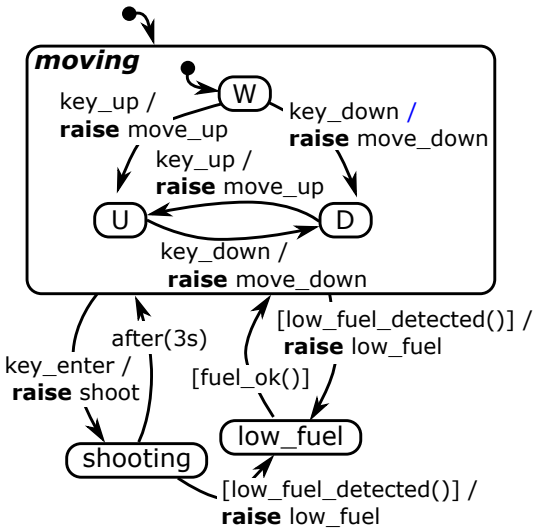

Fig. 6.3: A possible behavioural model of an example "tank wars" game.

earlier in this chapter, a more appropriate choice (to describe this timed, reactive, autonomous behaviour) is the *Statechart* language.

6.2.2 Discrete-Event Abstraction

Certain system behaviour, in particular the behaviour of control software, can be described using a discrete-event abstraction. In Figure 6.2 a view of the behaviour of an example system is shown – in this case, the system is a "tank wars" game, in which 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 fuelling 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 6.2, an input event *segment* is shown. A segment is a finite interval of time, in which a number of events occur. 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 behaviour 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 6.2. 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 middle of Figure 6.2. 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

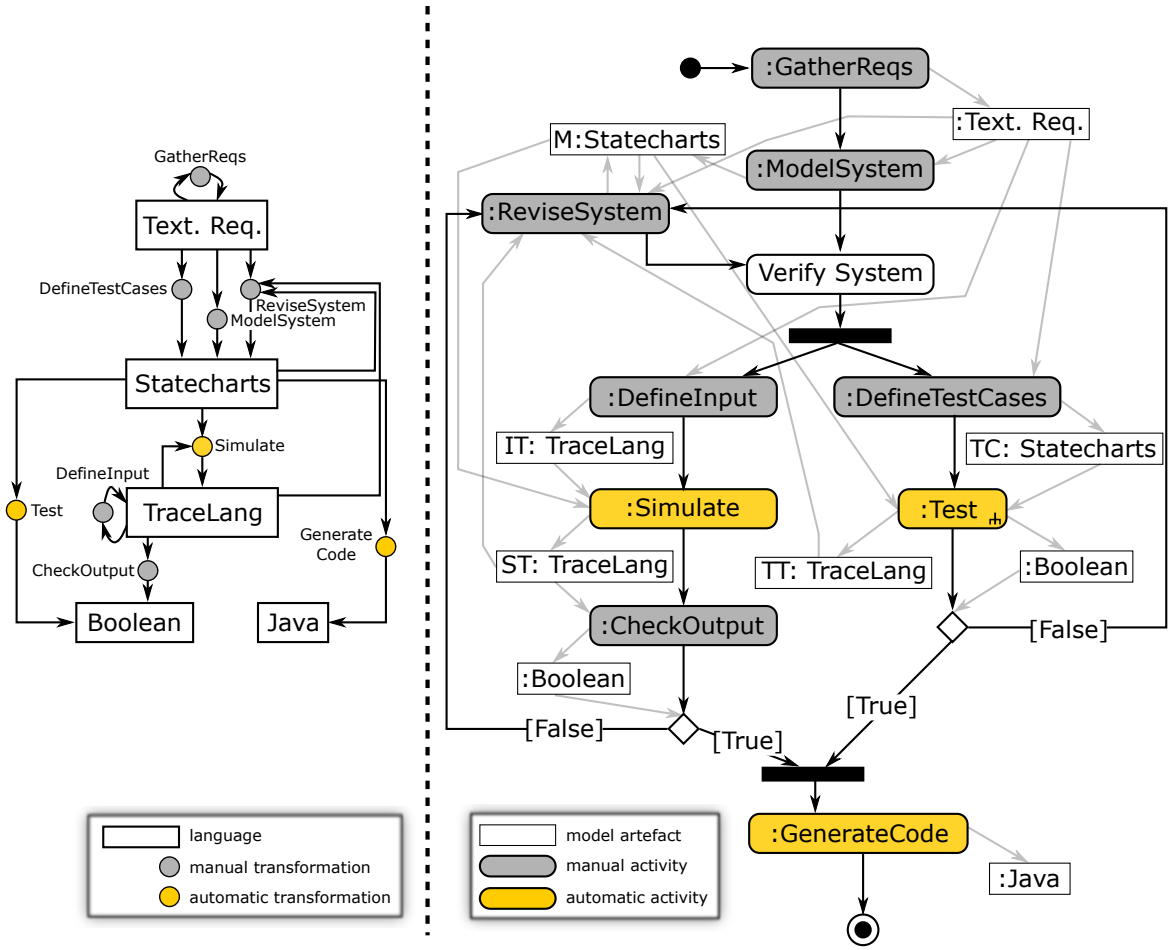

Fig. 6.4: The workflow for modelling, simulating, testing, and ultimately generating code from Statechart models.

system autonomously changes its state to *low_fuel*. From the input, output, and state trace, we can deduce that the system is *timed* (its behaviour includes timeouts), *reactive* (to input from the environment), and *autonomous* (as it can change the system state without any input from the environment).

To describe all possible state trajectories of the system, a state diagram can be used – see Figure 6.3 for a possible model describing the system's behaviour. 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 behaviour, including a possible diagrammatic notation. In the rest of this chapter, the *Statechart* formalism is explained as an example of such a diagrammatic language to describe the timed, reactive, and autonomous behaviour of systems.

6.2.3 Process

As a running example that demonstrates the use of Statecharts, we will develop a Statechart model describing the timed, autonomous, and reactive behaviour of a digital watch. Whenever a system is developed, however, it is important to consider which steps are taken and which artefacts are created in the system development process, and to describe this process in a model. This process, or workflow, will guide us throughout the chapter to design, simulate, test, and ultimately deploy our example system. Figure 6.4 shows a model of this process, in a *Formalism Transformation Graph and Process modelling* (FTG+PM) language [195].

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) is a map of all the *languages* used during system development. Each artefact produced in the process model conforms to a language in the formalism transformation graph. Moreover, it defines the *transformations* between the languages, which can either be manual or automatic. Again, there is a correspondence between activities in the process 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 (according to the discrete-event abstraction discussed earlier). This output trace is manually checked and a decision is made whether or not the 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 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. In our case, the system is deployed by generating Python code and coupling it to a visualization of the (simulated) digital watch as shown in Figure 6.1.

6.3 Modelling with Statecharts

The previous section introduces the requirements of our example digital watch example, explains how we can view its behaviour with a discrete-event abstraction, and describes a process for developing the system using the Statechart formalism. In this section, we explain the elements of the Statechart formalism that can be used to model the timed, reactive, and autonomous behaviour of systems. We start with the basic concepts of states and transitions, and gradually introduce the more advanced concepts of depth, orthogonality, broadcast communication, and history. Each new concept's syntax and semantics are explained, and is illustrated by progressively developing the running example's model.

6.3.1 States and Transitions

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 [150]. To illustrate the use of states and transitions, two parts of the digital watch behaviour are shown in Figure 6.5 and Figure 6.6, implementing the part of the system that updates the clock value every second, and the part of the system that is responsible for turning on and off the Indiglo light.

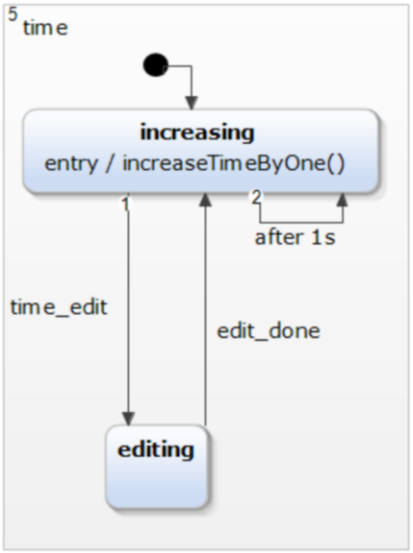

Fig. 6.5: Updating the clock value every second.

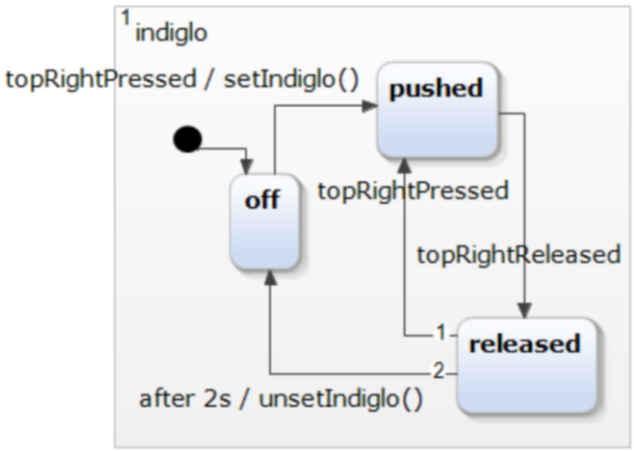

Fig. 6.6: Indiglo behaviour.

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 in the model is the initial state – on system start-up, the state of the system is initialised 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 6.6, three states are modelled, implementing the behaviour of the Indiglo light: *off* (the default state), *pushed*, and *released*. These names are strategically chosen to reflect the state of the actual ("physical") watch (the display). However, there is no direct link between the name of the state and the state of the display: the names are merely used to quickly convey their meaning to a person looking at the model; the actual behaviour (turning on and off the light) has to be implemented in the link between the Statechart model (or more precisely, its generated code) and the platform code (*i.e.*, the physical digital watch and its embedded software platform, or in our case, a simulated platform in Python). The state *off* represents the light being off; the state *pressed* represents the state where the user is pressing the top right button (turning on the light); the state *released* represents the state where the user has released the top right button.

Besides the state – or mode – the system is in, a system keeps track of a number of *state variables* $\{v_1, v_2, \ldots, v_n\}$. 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[1]. These variables with names $\{v_1, v_2, \ldots, v_n\}$ and types $\{t_1, t_2, \ldots, t_n\}$ have values $\{val_1, val_2, \ldots, val_n\}$ where $val_i \in dom(t_i)\ \forall i \in [1, \ldots, n]$

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 a number of conditions are 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) or it can be internal to the Statechart model. The triggering event can also be a

[1] https://www.itemis.com/en/yakindu/state-machine/documentation/user-guide/sclang_statechart_language_reference#sclang_types

timeout, which is identified by the reserved event name *after* and a parameter denoting the amount of time (using a certain time granularity) that will pass until the timeout triggers.
- A triggering *condition* (optional), which models a boolean condition on the state of the system. The condition can check the values of system variables and check whether a specific state (in another orthogonal component, see Section 6.3.3) is active.

A transition that has no trigger is said to be *spontaneous*. The transition leaving the marker for the initial state is always spontaneous. The transition's action can:

- Raise events, either local to the Statechart model, or to the environment. In general, an event has the following signature: $< event - name > (< event - params >)$.
- Perform computations and assignments on the system's variables.

In Figure 6.6, transitions are modelled that describe the dynamic behaviour of the Indiglo light: it turns on when the *topRightPressed* event arrives from the environment. It stays on as long as the button is not released; if it is, the light stays on for 2 seconds and then turns off again. The *interface* of this model consists of the set of accepted input events $X = \{topRightPressed, topRightReleased\}$, the set of possible output events $Y = \varnothing$, and a set of callback functions $\{setIndiglo, unsetIndiglo\}$. When this model is deployed (by generating code) and placed in an environment, the environment can send input events belonging to the input event set to the running system. It can listen to output events raised by the system and take appropriate actions. Moreover, on system start-up, it has to provide implementations for the call-back functions: these implementations are not known by the Statechart model, as they will change the ("physical") state of the system. In this sense, the same model can be placed in different environments to implement the same behaviour on different "physical" systems; for example, an updated version of the system's hardware might not necessarily entail a change in the behaviour of the system. In this case, we can update the hardware and its associated environment, but leave the Statechart model the same and generate code that is placed in the new environment.

6.3.2 Composite States

A composite state is a collection of substates, which themselves can be basic states or composite states. This allows for nesting states to arbitrary depths. The main purpose of composite states is to group behaviours that logically belong together. Transitions entering a composite state will enter its default state (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. It is also possible to enter a child state of the composite state directly by targeting it, of course. Transitions going out of a composite state can be thought of as being defined on all of the inner states as well – through a flattening procedure, it is possible to obtain an equivalent Statechart model that only contains basic states and transitions.

For example, in Figure 6.7, the behaviour of the alarm is modelled. A composite state *blinking* is defined: it is entered when the *checkTime* procedure returns true (signifying that the time of the alarm has been reached). Upon entering the *blinking* state, the default *on* state is entered as well (and, if an action is defined on the transition to the default state, it is executed as well). Two outgoing transitions are defined: when the user presses any button, or after 4 seconds, the alarm is turned off. Regardless of the internal active state of *blinking*, when the outer transition is triggered, that active state is exited before the *blinking* state is exited and the *off* state is entered.

One important issue with composite states is that unwanted non-determinism can occur if a substate has an outgoing transition that is triggered on the same event as its composite state's transition. 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. We can obtain the flattened version of a Statechart model by removing hierarchy: we only retain "leaf" (basic states), by adding the transitions that leave (or enter) their ancestor states on the leaf state. By doing this recursively (starting at the lowest level), we remove hierarchy by progressively adding these "higher-level" transitions to the basic states.

For example, in Figure 6.7, if there was a transition on the *topRightPressed* event from the state *on* to the state *off*, the model is non-deterministic in case the *on* state is active when a *topRightPressed* 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 [141] – or the inner-most transition can

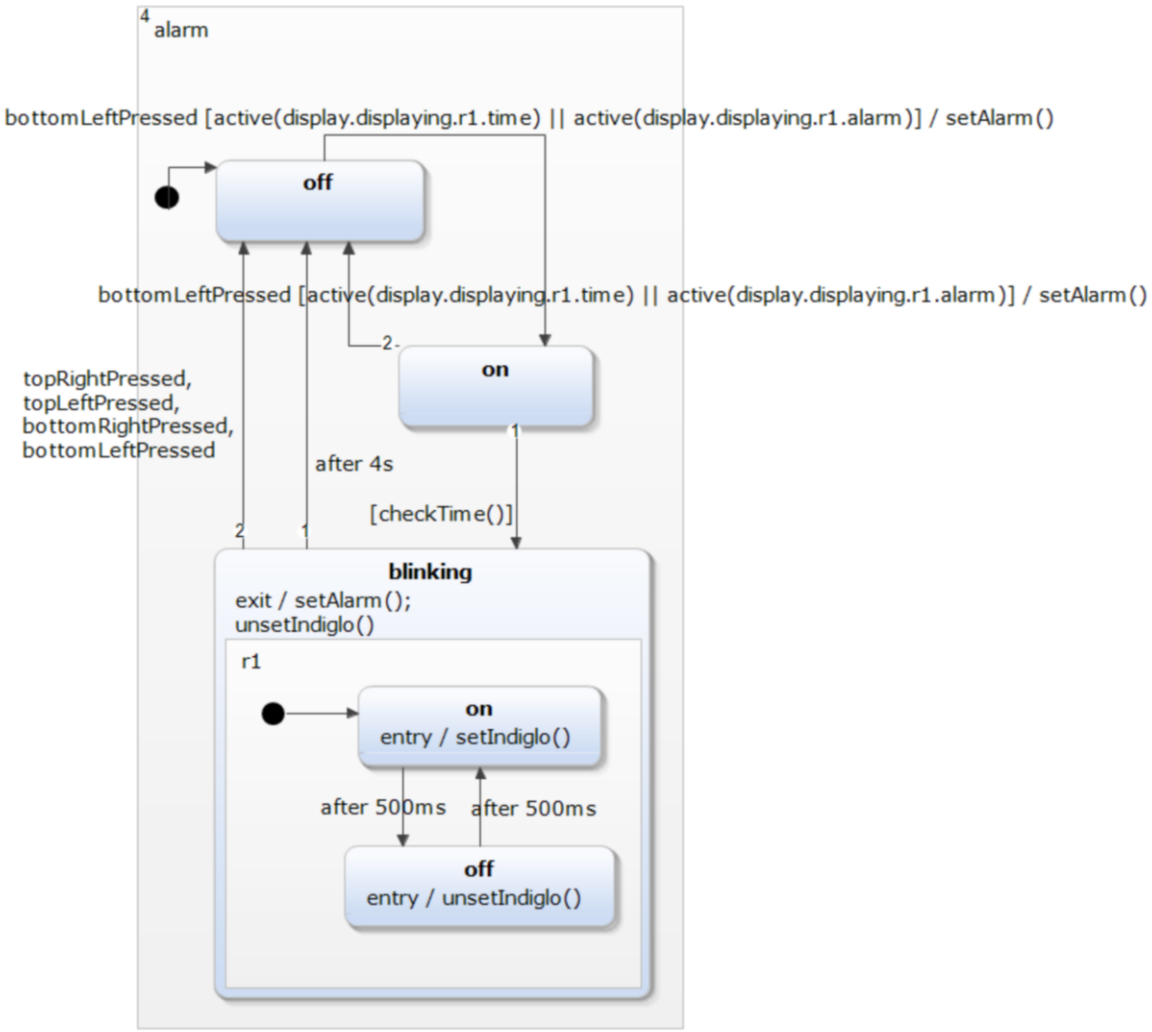

Fig. 6.7: The component that controls the alarm behaviour.

be chosen – as in Rhapsody [139]. These different options are presented in Figure 6.8, which also shows the flattened version of the models. In this tutorial, we assume STATEMATE semantics.

6.3.3 Orthogonal 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 which has concurrent regions, all regions execute simultaneously. Each region has a default state, that is entered when the region is entered. Orthogonal regions 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). A second way in which orthogonal regions can communicate is by querying the state of another orthogonal region: the triggering condition of a particular condition could include a constraint on which state is active in a different orthogonal region.

Figures 6.5, 6.6, and 6.7 all are orthogonal to each other. Indeed: the time needs to be updated, even if the Indiglo light is turned on, or the alarm goes off. Otherwise, the watch would be useless as a timekeeping device. A fourth orthogonal component is shown in Figure 6.9. It controls the display of the digital watch, by switching

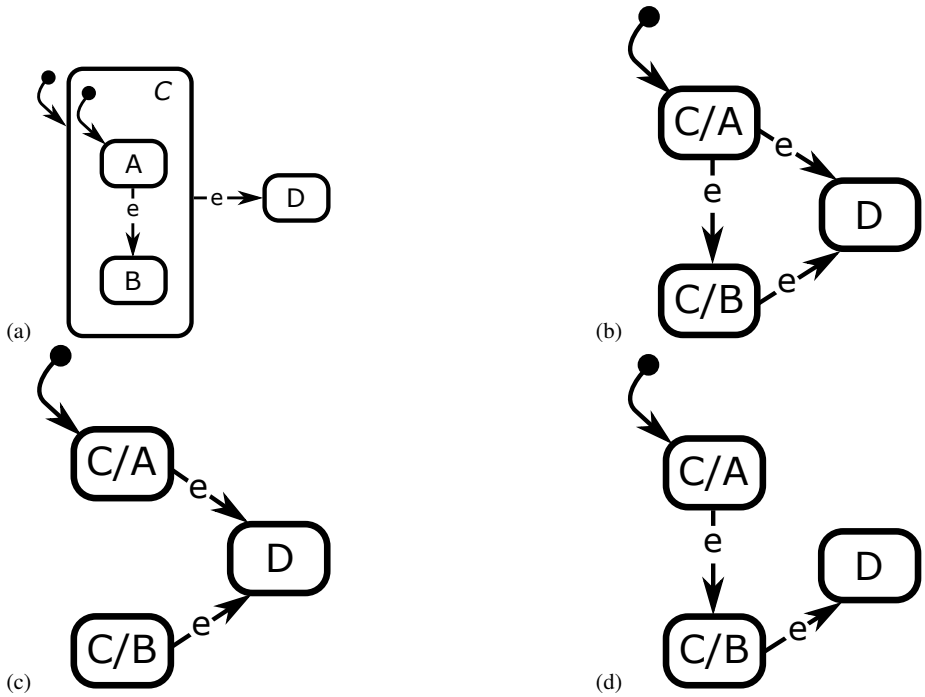

Fig. 6.8: Non-determinism in composite state; (a) an example model containing non-determinism, (b) flattened version: non-determinism in state A, (c) Statemate semantics: outermost transition is prioritized, (d) Rhapsody semantics: innermost transition is prioritized.

(in the *displaying* composite state) between the *displaying_time*, *displaying_chrono*, and *displaying_alarm* states when the user presses the top left button. The user can edit both the current time, as well as the time at which the alarm is set. For this, there is a transition from the *displaying_time* and *displaying_alarm* state to the *editing_time* state, where the user can increase the hours/minutes/seconds by pressing the bottom left button, and change the current selection by pressing the bottom right button. The *display* orthogonal component communicates with the *time* orthogonal component by raising the *time_edit* and *edit_done* events. This will ensure that the time is not updated while the user is editing it.

6.3.4 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 6.9, a shallow history state remembers the active direct child of the *displaying* composite state when it is exited (*i.e.*, when a transition to the *editing_time* state is taken). Upon returning from the *editing_time* state, the history state will restore the last active state, to ensure that we end up back in the *displaying_time* or the *displaying_alarm* state, depending on what we were editing. If no history state were present, we would always end up back in the *displaying_time* (default) state.

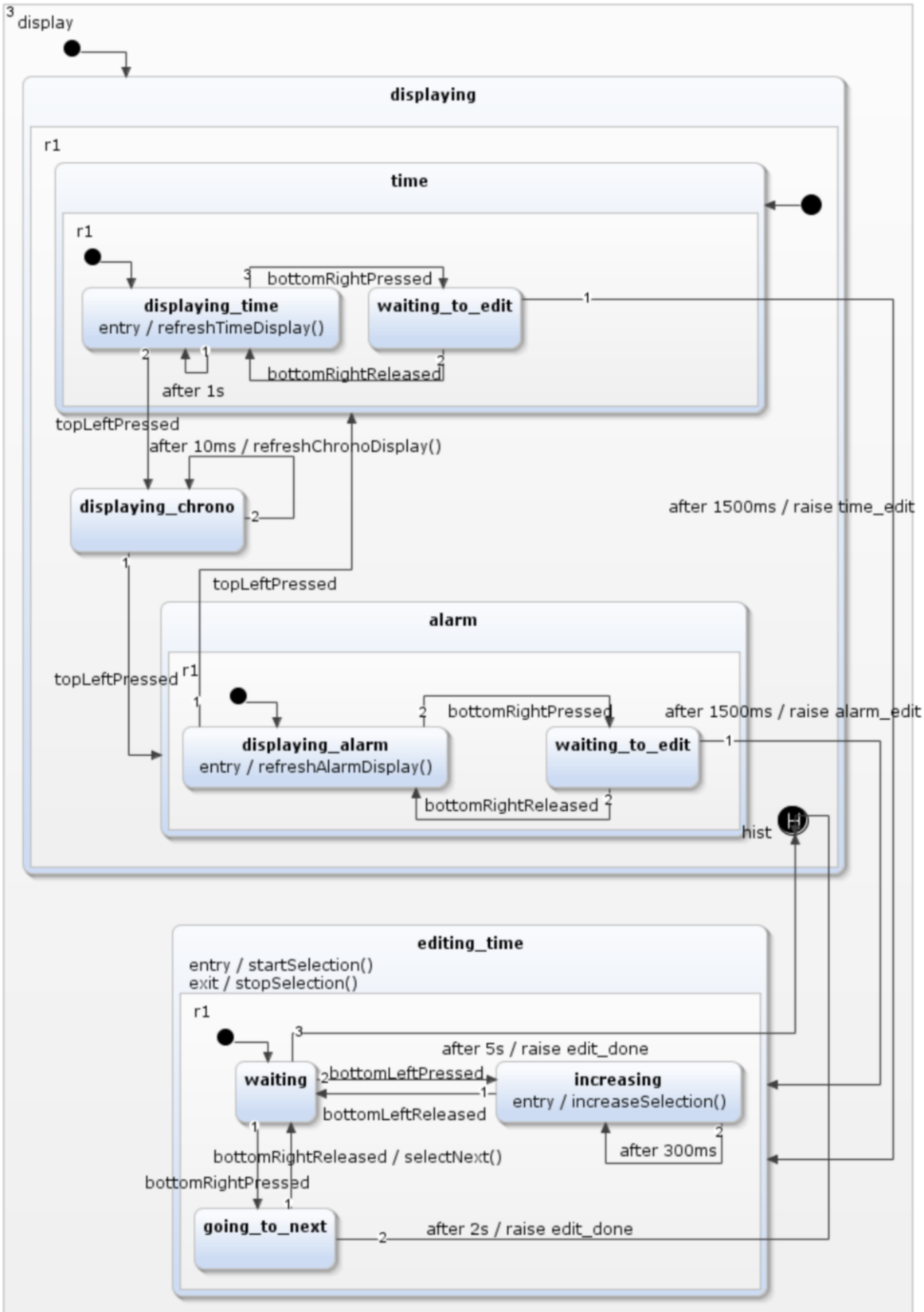

Fig. 6.9: The component that controls the display behaviour.

6.3.5 Syntactic Sugar

The previous subsections discussed the essential elements of the Statechart formalism. There are, however, additional syntactic constructs that make the modeller's life easier, but can be modelled 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 case if A and B are direct descendants of the same composite state. 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).

Entry and exit actions have been shown throughout the components of the digital watch model in this chapter. For example, for the alarm orthogonal component of Figure 6.7, whenever the *blinking* state is exited (which signifies that the alarm "triggered" and the blinking phase is over), the alarm is disabled by a call to *setAlarm*, and the Indiglo light is turned off by a call to *unsetIndiglo*. Were we not to model this action as an exit action, we would have to repeat it on the two transitions that leave the *blinking* state. Within the *blinking* composite state, the *on* state has an entry action that turns on the Indiglo light. Were we not to model this action as an entry action, we would have to repeat it on the incoming transition that marks this state as the initial state, and the incoming transition from the *off* state.

6.3.6 Full Statechart Model

The full Statechart model (where the previously discussed orthogonal regions have been placed in subdiagrams) of the digital watch is shown in Figure 6.10. One extra component not discussed in the previous subsections is the *chrono* component, which is responsible for increasing the chronometer when it is activated. Note that the chronometer can only be activated when the chronometer is shown (in the *display* component); the *active* function takes a state reference as a parameter and returns *true* when the state is active at that moment.

6.4 Detailed Semantics

The detailed semantics of Statecharts are more intricate than first meets the eye. In this section, we explain the semantics of Statecharts according to its implementation in STATEMATE [141]. As was already touched upon, some semantic variations are possible if multiple options can be chosen (for example, taking the inner or outer transition if there is a conflict in a composite state). In Section 6.7, we will explain that more options are possible.

The execution of a Statechart is divided into (micro)steps. An event queue holds the events that are currently raised (either by the environment, or locally by the Statechart). It also keeps track of timeouts that will expire in the future (corresponding to an *after* condition on a transition). In each microstep, the execution algorithm decides which transition(s) (in different orthogonal regions) are enabled. They are executed (in arbitrary order) by changing the state of the system from the source state of each transition to its target state. In detail, the execution algorithm works as follows:

1. A set of candidate transitions is generated that are enabled by events that were raised locally in a previous step, by timeouts, or by events coming from the environment. The candidate transitions are filtered by

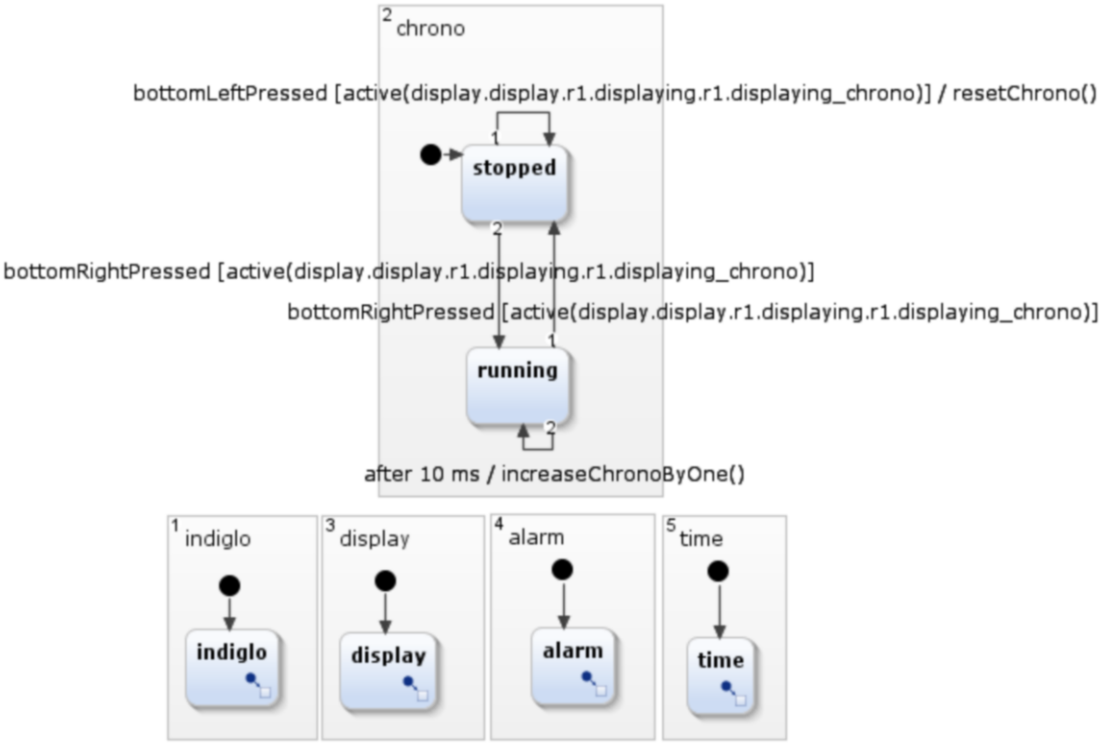

Fig. 6.10: The full Statechart model of the behaviour of the digital watch.

evaluating their condition (which is a boolean function on the total state of the system). The state of the system stays constant until all the transitions execute (at the same time); a function which checks whether a particular state is active does not depend on which transition fires first (as the check is done before any transition fires).

2. The set of candidates is checked for non-determinism (a conflict), of which there are two types:
   - Two candidate transitions can be in conflict if they have the same source state. In the case of STATEMATE, the execution algorithm cannot give priority to one or the other; as such, a warning is issued to the user and one of the candidates is chosen arbitrarily. More elaborate schemes could assign priority to these transitions based on other criteria; these will be discussed in Section 6.7.1.
   - Two candidate transitions can be in conflict if they are enabled in the same hierarchical state (on any level of the hierarchy). In case of STATEMATE, the top-most transition in the hierarchy is chosen.
3. Once the set of transitions to be executed is known, they are executed by following these steps for each of them (in arbitrary order, since we assume the orthogonal components are independent):
   a. The *Least Common Ancestor* (LCA) of the transition is calculated. This is the state that is lowest in the hierarchy of states that has both the source and the target state as descendant.
   b. The exit set of the transition is calculated. This set contains all the states that need to be exited when the transition is executed. This includes the source state of the transition, as well as all the states in the hierarchy up to (but excluding) the LCA.
   c. For each composite state in the exit set that has a history state as a direct child, the current configuration of the composite state is remembered.
   d. For each state in the exit set (in order, from lowest to highest in the hierarchy), its exit action is executed and the state is removed from the current configuration of the system.
   e. The action of the transition is executed. Actions that are modelled using code (such as value assignments) are immediately executed. Events that are raised are placed in the queue, either to be sent to the

environment as output at the end of the step, or to be sensed by the Statechart in the next iteration of the execution algorithm.

f. The effective target set of each transition is executed. This is a recursive procedure down to the "leaf" (basic) states. In case the target is a composite state, the target set is extended with the effective target set of its default state. In case the target is a state containing multiple orthogonal regions, the target set is extended with the effective target set of all its orthogonal regions. In case the target is a history state, the target set is extended with the effective target set of the remembered state(s).

g. All the targets in the effective target set are entered, as long as they are not an ancestor of the source of the transition. The targets are entered from highest in the hierarchy to lowest. Their enter action is executed and they are added to the current configuration of the system. For all the outgoing transitions of the entered states that have a timeout, an event is scheduled in the future.

4. All events that were raised during the step are processed: they are either sent to the environment as output events, or (in case they are local) used in the next microstep.

This process is continued indefinitely – or until an end condition on the execution is satisfied. For example, a particular state could syntactically be designated as the "final" state, or a condition could be defined on the full state of the system. In any case, the execution algorithm is responsible for respecting the timings specified in *after*-events, by pacing the execution in such a way to approach real-time execution (with or without guarantees, depending on the operating system).

6.5 Testing Statecharts

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 6.4. 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 behaviour of the generator and acceptor, as they are timed, autonomous, reactive systems. This is illustrated in Figure 6.11, where we develop a test case for (a part of) the digital watch model. The generator and acceptor are modelled as orthogonal regions alongside the actual system.

The test case tests the expected behaviour of the digital watch in case the user wants to edit the time. To do this, an input event corresponding to the user pressing the bottom right button is raised after 1 second. The acceptor checks whether this results in the system switching to the *waiting_to_edit* state. Next, after an additional 1.6 seconds, the bottom right button is released, which should result in the system state changing to *editing_time*. Then, 0.4 seconds later, the bottom left button is pressed; this results in the *increasing* state to be activated until the bottom left button is released (after 3 seconds). Then, 5 seconds later, the system should automatically switch to the *displaying_time* state again. If the test ends up in the *passed* state, the *testPassed* callback is called, signifying that the test has passed. Otherwise, the *testFailed* callback is called.

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 modelled 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 chapter.

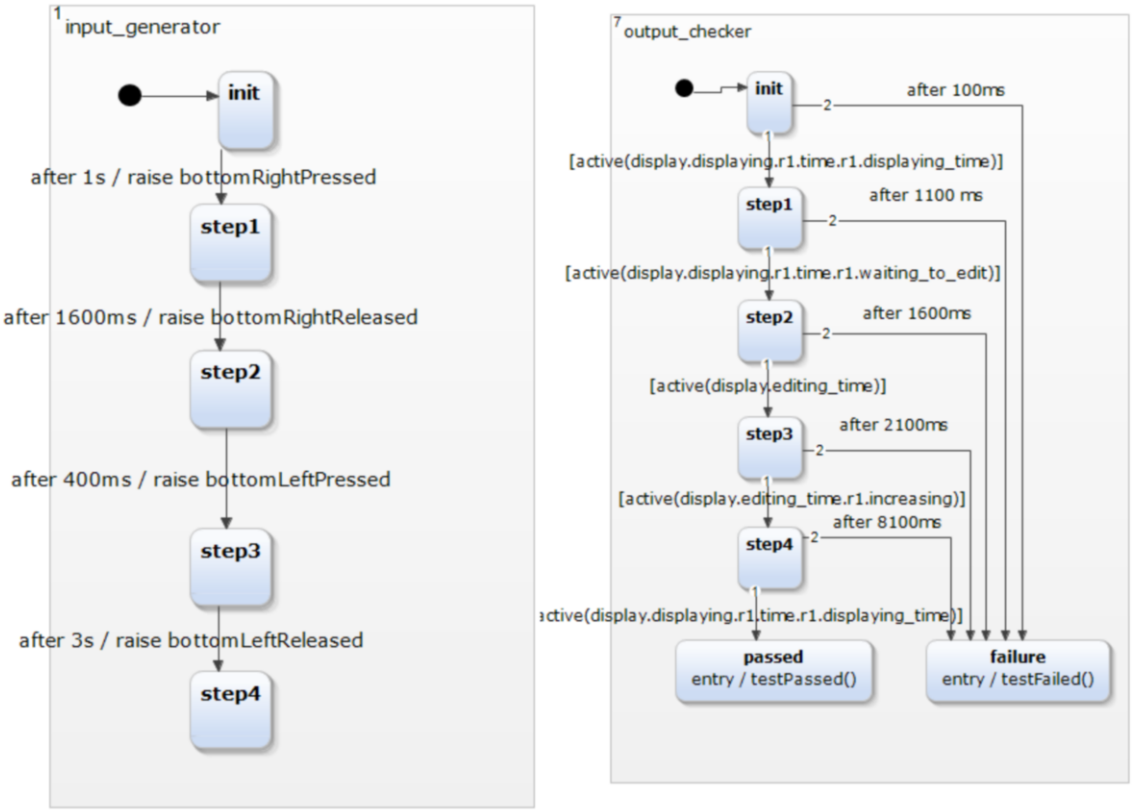

Fig. 6.11: The extra components of the digital watch test.

6.6 Deploying Statecharts

Once we have specified the timed, reactive, autonomous behaviour of a system, it needs to be deployed onto a target platform. The most common method of doing so is by generating code in a high-level language such as C++, Python, or Java. The generated code is then coupled (through its interface of input/output events and callback operations) to a (hand-coded) platform that implements the non-modal (*i.e.*, non-timed, -reactive, or -autonomous) behaviour. Most often, this platform consists of a number of library functions that perform some action, such as displaying information on a screen, sending a signal to an actuator, or querying the value of a sensor. An execution kernel for the platform also has to exist, which is responsible for implementing the behaviour specified by the Statechart – more information on these platforms can be found in Section 6.7.2.

For code generation, we rely on Yakindu. Yakindu is a Statecharts modelling and simulation tool, with the following features:

- A graphical modelling 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 behaviour. 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 6.3.6 shows the digital watch model, as it was modelled 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 callbacks. 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 modeller can make. We can, from this model, also generate a running application. In our case, we will generate the behaviour and visualize it in a Python GUI, which shows the current state of the system (depending on the display mode either the current time, the chrono, or the alarm, as well as the Indiglo light), and allows to interact with the system by pressing the four buttons of the watch. To do this, we define a visualization library that can display the state of our system. It has the following interface:

- *refreshTimeDisplay()*: displays the current value of the clock.
- *refreshChronoDisplay()*: displays the current value of the chronometer.
- *increaseTimeByOne()*: increases the value of the clock by one second.
- *setIndiglo()*: turns the Indiglo light on.
- *unsetIndiglo()*: turns the Indiglo light off.
- *resetChrono()*: resets the chronometer.
- *increaseChronoByOne()*: increases the chronometer by $\frac{1}{10}$ seconds.
- *startSelection()*: starts the selection for editing the time (or alarm).
- *stopSelection()*: stops the selection for editing the time (or alarm).
- *selectNext()*: selects the next element (hours/minutes/seconds) to edit.
- *increaseSelection()*: increases the current selected element (hours/minutes/seconds).
- *setAlarm()*: toggles the alarm.
- *checkTime(): boolean*: returns true when the alarm time is reached.
- *refreshAlarmDisplay()*: shows the set alarm time.
- *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 (and change) the current state of the digital watch, as is shown in Figure 6.1, where the current time is shown (corresponding to the default display). To connect this GUI to the code generated by Yakindu from the Statechart model, we define appropriate listeners for the buttons in the interface: pressing/releasing the bottom left/top left/bottom right/top right button will send an appropriately named event from the set *{bottomLeftPressed, bottomLeftReleased, topLeftPressed, topLeftReleased, bottomRightPressed, bottomRightReleased, topRightPressed, topRightReleased}*.

The callback functions that are called during the execution of the Statechart are implemented by calling the function with the same name in the interface. This coupling is trivial, but more complex couplings are possible, as long as the callback functions themselves do not introduce waiting behaviour, spawn threads, or take too long to execute.

This development method allows for cleanly separating behaviour (encoded in the model, and generated to executable code by an appropriate code generator) and the presentation (encoded in a visualisation 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.

6.7 Advanced Topics

This section explores a number of advanced topics related to Statecharts. We first explore possible semantic variation for Statecharts; we can consider Statecharts not as a single formalism, but a family of formalisms, whose semantics differ slightly. Then, we explain that Statecharts can be executed (deployed) on multiple possible platforms, which has an effect on how the kernel handles the event queue and how time is advanced. Last, we explore ongoing work to introduce dynamic structure to allow for multiple concurrent objects ("agents").

6.7.1 Semantic Variations

In Section 6.4, we've explained the execution semantics of Statecharts with *small steps*: an (unordered) execution of an enabled transition in each orthogonal region. While this is a correct specification, we can ask ourselves several questions:

- When are events that are raised by a transition visible and usable to calculate a new set of enabled transitions?
- How is conflict resolution performed if there are multiple transitions enabled within the same orthogonal region?
- When do we process an external event? Immediately after a small step, or do we wait until the system is in a *quiescent* (*i.e.*, no transitions are enabled) state?
- When are assignments to variables visible? Immediately after a transition is executed, after a small step, or even later?

In their paper, Esmaeilsabzali et al. explore these semantic options for big-step modelling languages (a set of languages that Statecharts belongs to) [94]. These semantic options lead to a *family* of (related) languages: there is no longer *the* Statechart language, but one can configure the language in such a way to suit the application at hand.

At the essence lies the observation that there are three *layers* of steps: small steps, combo steps, and big steps. A big step is a sequence of combo steps, and a combo step is a sequence of small steps. The special property of a big step is that after it ends, environmental input (events sent to the Statechart) and output (events sent to the environment) are processed. One could state that small steps and combo steps are *internal* steps, while big steps lead to *externally visible* changes. A small step behaves the same as defined before, however, the choice can be made to not allow concurrency in a small step: only one transition is executed in that case. Moreover, when concurrency is allowed, a choice can be made whether transitions can *preempt* each other. A combo step groups a sequence of small steps. In the following paragraphs, we explore a number of semantic choices that can be made; these choices configure the Statechart language, resulting in a language "variant".

Big Step Maximality

This semantic choice decides when a big step ends, which always has to be in a "consistent state" (so after at least one small step). One option (*syntactic*) is to designate certain states as "stable" control states; in that case, a big step ends after a series of small steps that end in such a state. A second option is *take one*, which executes exactly one small step. And last, there is the *take many* option, which executes small steps until there are no more enabled transitions.

In the first and last options, a downside is that big-steps might not terminate (either because a designated control state was not entered or because transitions remain enabled). A downside of the second option is the fact that big steps have no clear syntactical scope (while in the other cases the scope of a big step can be deduced from the model).

Combo Step Maximality

This semantic choice is identical to the big step maximality, but then applied to combo steps. The main difference between combo steps and big steps is that after a combo step, no environmental input can be processed.

Small Step Concurrency

While the big step maximality and combo step maximality govern how many small steps can be executed, this semantic option allows for one small step to consist of a *single* transition execution or *many* (concurrent, enabled) transition executions. The advantage of the *single* approach is its simplicity, but it requires that the transitions are ordered non-deterministically. The *many* approach, on the other hand, has the potential for race conditions.

Small Step Preemption

If one transition is an interrupt for another transition (this is the case when both transitions are enabled in orthogonal regions, and one of them has a target state which is not orthogonal to the source states of both transitions, which effectively exits both orthogonal regions), and *many* concurrency is enabled, this semantic option allows to control what happens. Either both transitions are executed (*non-preemptive*, the result being the execution of both the transition's actions, and the target state outside of the orthogonal regions to be active), or only the interrupting transition (*preemptive*) is.

Event Lifeline

The *event lifeline* specifies how long an event "lives" (and can trigger a transition). Both input events and internally generated events have a lifeline (which can be different). There are five options:

1. With *present in whole*, the event is assumed to be present throughout the full duration of a big step. This leads to counterintuitive behaviour (since this might lead to non-causal behaviour, where an event generated by a transition "later" in the big step might cause a transition "before").
2. With *present in remainder*, the event is available in the remainder of the big step. This is an intuitive option, but can lead to unwanted non-determinism: depending on the order in which transitions are executed, a transition can generate an event and immediately enable a transition which would be disabled in a different ordering.
3. With *present in next combo step*, a generated event can only be sensed in the next combo step. This ensures that the ordering of transitions within a combo step cannot affect a transition being enabled or not. However, the ordering is only partial (since combo steps are potentially ordered non-deterministically).
4. With *present in next small step*, a generated event can only be sensed in the next small step. The ordering of transitions within a small step, nor the ordering of combo steps can affect whether a transition is executed in a particular run.

This semantic option has a potentially major impact on the execution of a Statechart model. It also shows the potential benefits of splitting up the executions in big steps, combo steps, and small steps.

Transition Priority

The last semantic option resolves the issue when two or more sets of transitions exist that can be executed in a small step. In that case, a choice has to be made which set is to be executed. A common example is when two transitions in the same state hierarchy are enabled. The *hierarchical* choice lets the priority be determined by the source and destination states of the transitions; for example, in STATEMATE [141], the outermost transition is chosen, while in Rhapsody [139], the innermost transition is chosen. More elaborate schemes are possibly within this choice, however. The *explicit priority* option lets the modeller assign priorities explicitly in the model. Last, the *negation of triggers* requires the modeller to go through all transitions and strengthen the event triggers in such a way that conflicts are avoided. The last two options are tedious to use, but allow for precise control over the priorities.

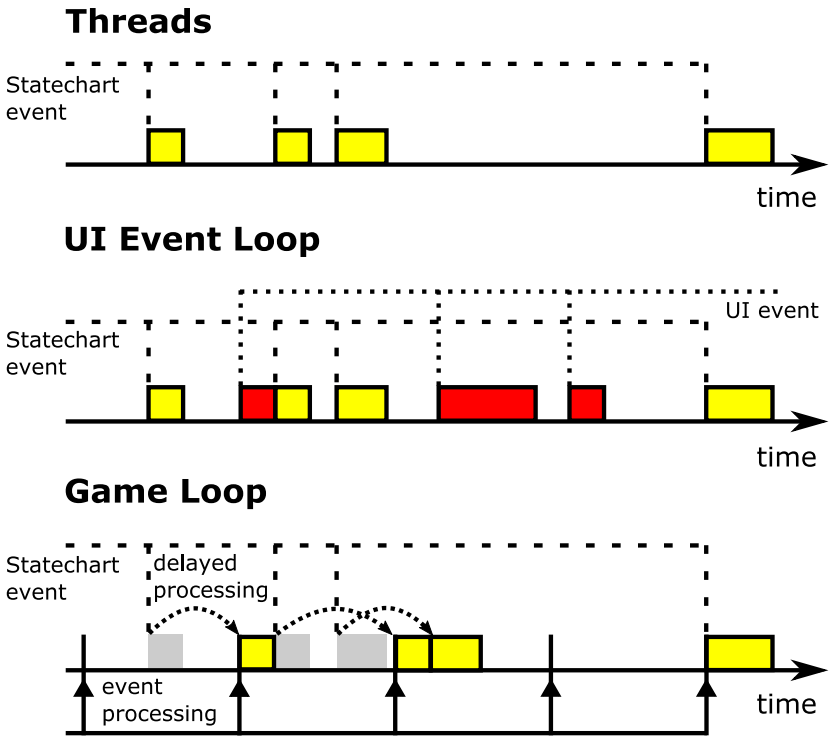

Fig. 6.12: The three runtime platforms.

6.7.2 Execution Platforms

We have discussed deployment of Statechart models in Section 6.6. In particular, we discussed how to provide an implementation for the computation functions that are called in the actions of the Statechart model, and how a "glue" layer has to be constructed that correctly sends input events to the Statechart and interprets output events coming from the Statechart. We did not yet consider *who* the control in the deployed application has; we always consider it is the Statechart. But often, Statechart models are embedded in applications that also require to have control, such as UI systems and game engines. This section explores how the Statechart kernel can work on these platforms by relinquishing certain key control functions.

The semantics of Statecharts (configured according to the semantic options presented in the previous section) are executed on a runtime platform, which provides essential functions used by the runtime kernel, such as the scheduling of (timed) events. The kernel attempts to run the Statechart model in real-time, meaning that the delay on timed transitions is interpreted as an amount of seconds. The raising of events and untimed transitions are executed as fast as possible. It is useful to discuss these platforms, as the Statechart simulators/executors depend on them; moreover, we need different types of platforms for different application scenarios. We distinguish three platforms, each applicable in a different situation. Figure 6.12 presents an overview of the three platforms, and how they handle events.

The most basic platform, available in most programming languages, is based on threads. In its simplest form, the platform runs one thread, which manipulates a global event queue, made thread-safe by locks. Input from the environment is handled by obtaining this lock, which the kernel releases after every (big) step of the execution algorithm. This allows external input to be interleaved with internally raised events. Running an application on this platform, however, can interfere with other scheduling mechanisms (*e.g.*, a UI module), or with code that is not thread-safe.

To overcome this interference problem, the event loop platform reuses the event queue managed by an existing UI platform, such as Tkinter. The UI platform provides functions for managing time-outs (for timed events), as well as pushing and popping items from the queue. This results in a seamless integration of both Statechart events and external UI events, such as user clicks: the UI platform is now responsible for the correct interleaving.

The interleaving is implemented in the "main loop" of the UI platform, instead of (for the threads platform) an infinite *while*-loop.

The game loop platform facilitates integration with game engines (such as the open-source Unity[2] engine), where game objects are updated only at predefined points in time, decided upon by the game engine. In the "update" function, the kernel is responsible for checking the current time (as some time has passed since the last call to the "update" function), and processing all generated events. This means that events generated in between two of these points are not processed immediately, but queued until the next processing time.

6.7.3 Dynamic Structure

The exclusive use of Statecharts does not scale to the complex (and often dynamic-structure) behaviour of complex software systems, such as (WIMP) graphical user interfaces, games, multi-agent systems, ... Their complexity goes further than the timed, reactive, autonomous behaviour Statecharts is good at representing. In many applications, there is a need for a set of collaborating, concurrent objects, that can appear and disappear at runtime. Object-oriented modelling methodologies address software complexity and allow for the construction of a set of interacting objects, but are not specifically designed for modelling timed, interactive discrete-event systems [137].

A number of approaches, such as OO Statecharts [137] were proposed. More recently, SCCD [281] was presented, combining the structural object-oriented expressiveness of Class Diagrams with the behavioural discrete-event characteristics of Statecharts.

Figure 6.13 presents an overview of the SCCD language, applied to a "bouncing ball" application: the application allows for a user to create a number of balls on the screen. Their autonomous behaviour is simply to move in a certain direction and bounce off the sides of the screen. The user can select and delete balls. As we can see, a number of objects are present: an instance of the *Window* class, and two instances of the *Ball* class are shown. There are a number of links between objects; these links can be used to send or receive events. The object diagram (consisting of objects and links) *conforms* to the class diagram part of the SCCD model. On the other hand, each object has a runtime state, corresponding to one of the states in its Statechart model. A central entity, the object manager, allows for instances to request a new object or link to be created or deleted.

Objects can communicate using events: SCCD specifies a number of event *scopes*. A *narrow cast* allows for an object to send an event over a connection to another specific (set of) object(s). A *broad cast* will send the event to all other objects in the application. An *output event* works similarly to an output events in Statecharts: it is sent to the environment of the application. A special *cd* scope is defined for sending event to the object manager.

In essence, the objects of a Statechart run concurrently and communicate in a more "loose" way than the orthogonal regions of a Statechart. The Statechart model in each object has to finish a big step before events can be sent to other objects or input from other events can be processed. The communication between objects is asynchronous: they are not able to call each other's methods, or change their variable values directly. In this sense, SCCD can be seen as a semantic basis for multi-agent systems, where agents have a well-defined interface through which they can communicate.

6.8 Summary

This chapter demonstrates how to use the Statechart formalism to model the timed, reactive, autonomous behaviour of a system. The chapter is built around a development process that starts from requirements gathering, to build an initial design of the system. The concepts of the Statechart formalism (states, transitions, hierarchy, orthogonality, broadcast communication, and history) are explained by a running example of a digital watch. This design is refined by simulating the model and testing it, by providing an input event segment to the simulation/test and checking the output produced by the model (either manual in a simulation, or automated using an oracle in a test). Ultimately, code is generated from the design model and deployed on a platform.

[2] https://unity3d.com/

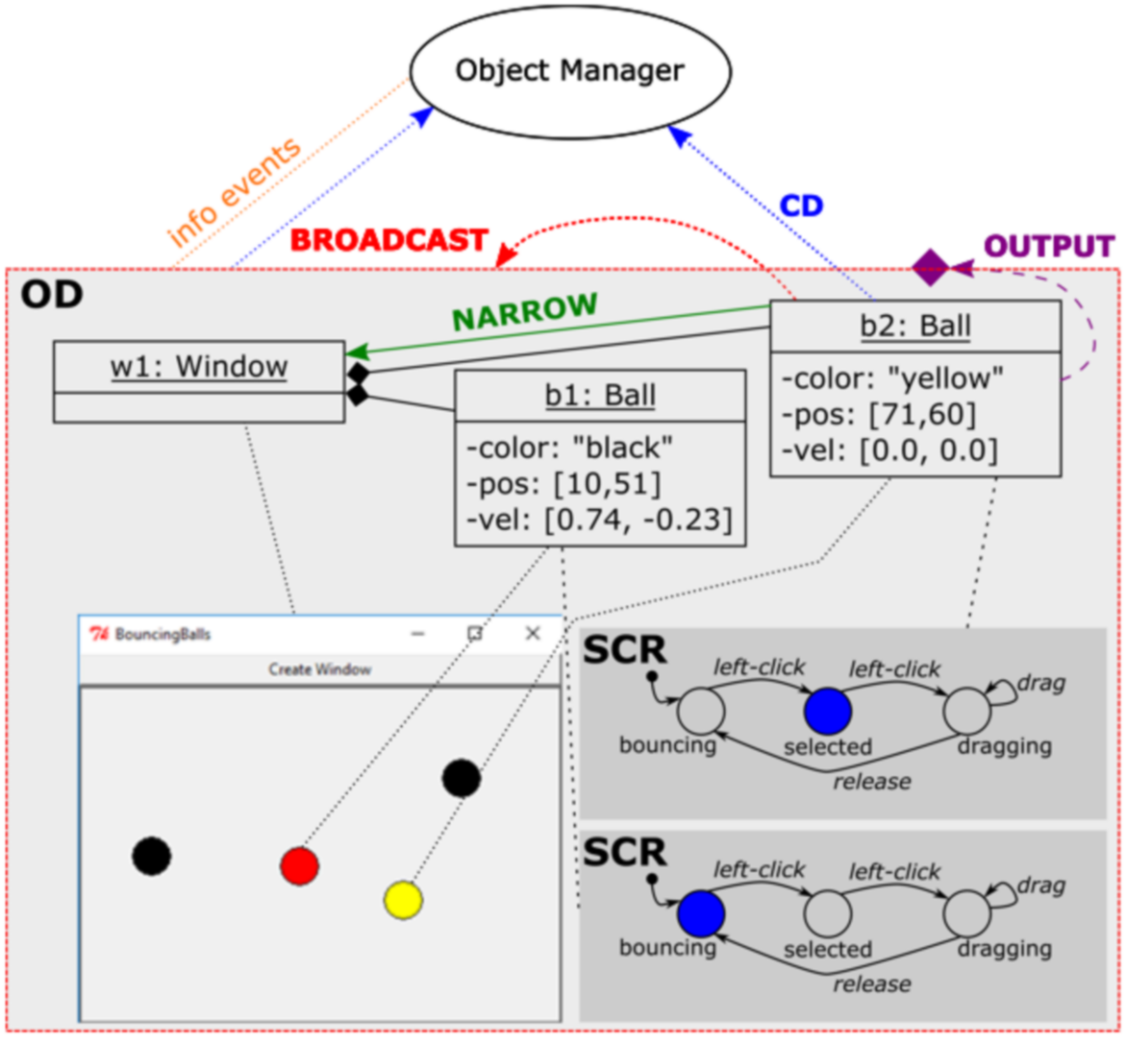

Fig. 6.13: SCCD: An overview.

Coupling between the platform functions and the Statechart output events/callback functions is required to finally arrive at a running system. Last, a number of advanced topics were discussed.

6.9 Literature and Further Reading

A thorough discussion of the origins of the Statecharts visual notation can be found in [136]. The origins of the Statecharts formalism are described in [134]; this work was later complemented with a discussion of its semantics in STATEMATE [141] and Rhapsody [139]. More recently, Esmaeilsabzali et al. have deconstructed the semantics of "big-step modelling languages", a family of languages to which the Statecharts language belongs, and have presented a family of their execution semantics [94]. A standard for the precise execution semantics of UML state machines is being worked on by the OMG [221].

Coordinating concurrently executing Statechart objects has its roots in a 1997 paper by Harel and Gery [137]. SCCD [281] attempt to constrain the possible object operations (creation and deletion of objects, as well as communication between objects) by modelling the allowed object configurations in a class diagram, to which each object diagram (a snapshot in the execution of the model) has to conform.

6.10 Self Assessment

1. Explain the difference between deep history and shallow history.
2. Explain the different ways in which Statechart variants (in particular, Statemate vs. UML 2.0/Rhapsody) resolve non-determinism when outgoing transitions of hierarchically nested states are triggered by the same event.
3. Can a Statechart model ever "block"? Explain why/why not.
4. Draw the flattened version (no hierarchy or orthogonality) of a Statechart model containing hierarchical states and orthogonal components.
5. How are transitions outgoing/incoming from/to a hierarchical state containing orthogonal components executed? Explain the different ways in which orthogonal regions can "interact". Explain the difference between concurrently running objects in SCCD and orthogonal regions within these objects.

Acknowledgements

This work was partly funded by Flanders Make vzw, the strategic research centre for the manufacturing industry.