RAINER KLUTE

HOW YOU CAN

MODEL YOUR SOFTWARE

USING

STATE MACHINES

YAKINDU

Statechart Tools

JUNE 2017

Table of contents

01. Introduction

Knowledge of finite-state automata is one of the foundations of theoretical computer science. Aside from all theory, this paper provides a practice-oriented entry into finite-state automata or state machines, which are far more than a mere theoretical construct. State machines are extremely useful in practical work. In this white paper we will show the sort of tasks for which state machines are suitable. Using examples, we will introduce both basic and some advanced concepts of state automata, demonstrating them using YAKINDU Statechart Tools [1].

Let's first take a look at some concepts for implementing finite-state automata.

02. Models in computer science

Computer science is about solving specific tasks with the help of computers or computer-controlled systems. Examples might include the creation of a railway timetable, the visualization of neutron fluxes and distribution of power in a nuclear reactor, the simulation of physical processes in semiconductor components, the implementation of a social media app or the development of machine control systems of various types. Last but not least, numerous control programs can be found in embedded devices.
A model of what is to be implemented in software should be available before programming begins. A model is a simplified representation of reality, and excludes everything that is irrelevant to the task to be solved. In the control system of a coffee machine, for example, the fill level of the beans is important, while the colour of the coffee cup or the gender of the operator is not.

The model should be clear and easy to understand, and through meaningful structuring should master even complex interrelationships. Comprehensible models are important, not only for software architects and developers, but also for their clients, i.e. departments and customers. If a model is comprehensible, the client can use it to determine whether the IT people have understood the task correctly. Mistakes and misunderstandings can be clarified before any effort is wasted in implementing an incorrect solution.

Certain tasks can be described particularly well with the aid of state machines [2]. Computer science students may get to know state machines as deterministic automata, finite-state machines, Moore-, Mealy-, Harel machines or the like in theoretical computer science – but not infrequently question marks remain on their faces, as if to say: "What is that supposed to be?"

03. Finite state machines: no mere theory

State machines are suitable for describing so-called "event-discrete systems". Such a system is always in exactly one of a finite number of states. For example, a light switch is always in one of two states, „On" or „Off". State transitions define from which state other states can be reached and under which conditions each state transition occurs.

Before this discussion becomes theoretical, let's have a look at a concrete example: a control system for a blind. In the simplest case, the blind has three states:

- The blind is not moving (state **Idle**).
- The blind is moving upwards (state **Moving up**).
- The blind is moving downwards (state **Moving down**).

In a state diagram, states are represented as rectangles with rounded corners, and state transitions as arrows, as shown in Fig. 1:

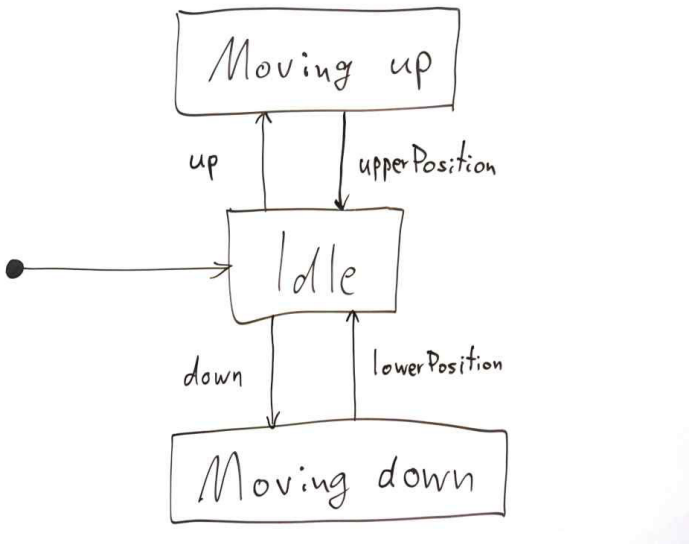

Fig. 1: State diagram of a blind control system

The point on the left indicates the initial state of the automaton, **Idle**. If the user presses the [↑] key, the *up* event occurs, the automaton changes to the state **Moving up**, and the blind moves upwards. As soon as it reaches the upper position, the position sensor generates the *upperPosition* event. This triggers the transition from **Moving up** to **Idle**. Closing the blind works analogously when the user presses the [↓] button. The state diagram illustrates the dynamic behaviour of the system.

The model, however, is still imperfect. We want to be able to position blinds not only fully open or fully closed, but also at intermediate positions as required. In addition, the blind should automatically close when there is strong sunlight. However, it ought to remain possible to control the blind manually, overriding the automatic operation.

In addition the blind should always be raised in strong wind to avoid damage. Users should not be able to override this feature: it should not be possible to lower the blind until the wind has decreased. It may also be required to raise and lower the blinds at specific times.

04. Modelling state automata using the blind control as an example

We want to model some of these functions with the help of a state machine. However, rather than using a pencil and a whiteboard, we want to deploy a capable modelling tool. In the following example we used YAKINDU Statechart Tools. It is Eclipse-based and supports not only modelling itself, but also dynamic simulation of the model and generating the state machine as source code in various programming languages. This is discussed later.

First we will convert the whiteboard model point by point using YAKINDU Statechart Tools. If you want to try this for yourself, you can download the software. For quick access, you will find an installation guide [3] and a five-minutes tutorial [4] on our website.

Modelled with the tool our blind control example looks like this:

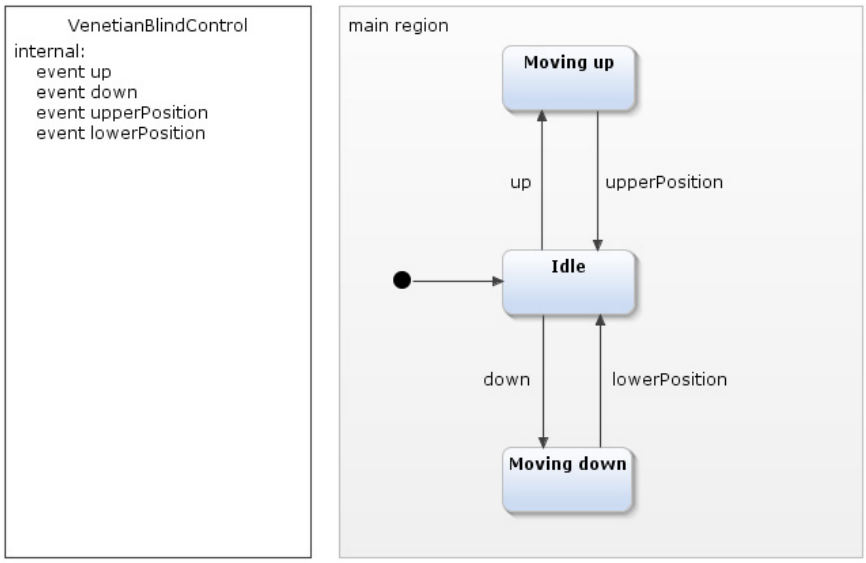

Fig. 2: A simple blind control system modelled with YAKINDU Statechart Tools

The definition section, which defines the names of events, has been added on the left-hand side.

To stop the blind on its way up or down at the current position, the user presses the button that is opposite to the respective movement, for example, [↓] when the blind is moving up.

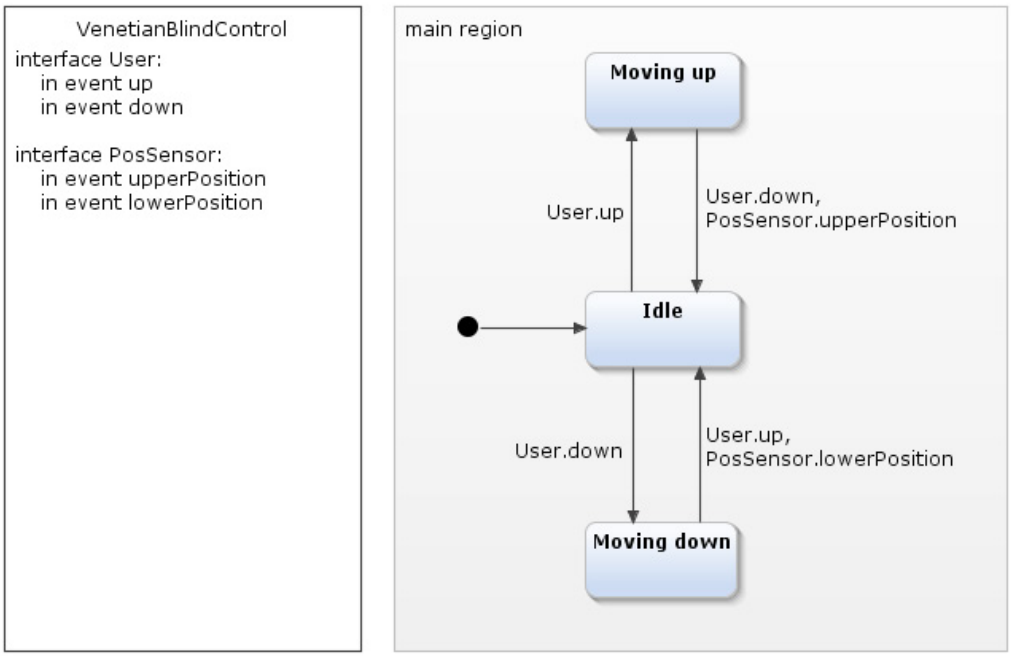

Fig. 3: Model of a blind control system: position sensors and events

The corresponding extended model looks like this:

There are now two possible events instead of one, which lead from of the states of movement into the rest state. Their names are next to the transitions, separated by commas.

At the same time we have made a clear distinction between events that are triggered by the user and those triggered by the position sensor of the blind. YAKINDU Statechart Tools refers to these different scopes as *interfaces*. The definition section determines which interfaces exist and which events are assigned to them. In our case this is the *User* interface with the events *up* and *down*, as well as the interface *PosSensor* with the events *upperPosition* and *lowerPosition*.

04.1 Conditional state transitions with variables and guard conditions

Let's now introduce prevention of storm damage to the blind. For this purpose the blind has a wind speed sensor that provides values between 0 (no wind) and 100 (the maximum value of the sensor).

In the state machine the wind speed is available in the *windspeed* variable of the *Wind* interface. To keep the example simple we will not go into how the wind speed is transferred to the variable. The *Wind* interface also contains the constant *STRONG*, which indicates the speed at which the wind is strong enough for the blind to be raised.

There is now an additional transition (blue arrow) from the state **Moving down** to the state **Idle**. This is provided with a condition, „Wind.windspeed >= Wind.STRONG". This so-called *guard condition* is enclosed in square brackets, and ensures that the transition is executed only if the condition is met. A further transition leads from **Idle** to the state **Moving up**, but only if the blind – as determined by the position sensor – is not already up. To make this information available the *PosSensor* interface is extended accor-

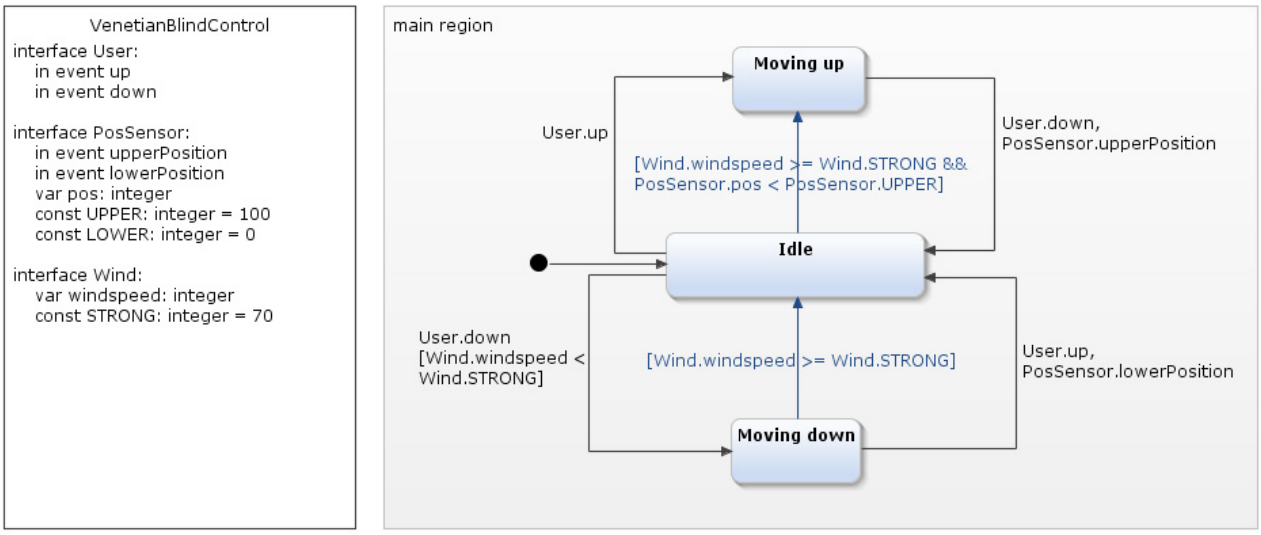

Fig. 4: Model of a blind control system with wind speed sensing

dingly.

The „blue" transitions in the above state diagram thus ensure automatic raising of the blind. Now we just have to prevent the user from moving the blind down again. On the transition **Idle → Moving down** the guard condition „Wind.windspeed < Wind.STRONG" assigned to the *User.down* event ensures this.

For the transition **Moving up → Idle**, however, it is not quite that simple, as it should be executed at the *PosSensor.upperPosition* event in any case. However, a guard condition always refers to all the events mentioned, so it would not be useful here. The correct solution is to split the transition into two individual transitions: one transition takes care of the event *User.down* and contains the guard, the other leads to the **Idle** state without any additional condition when the *PosSensor.upperPosition* event occurs.

04.2 Time-controlled triggering of state transitions

We have looked at the basic elements of state machine modelling; let's now look at other concepts and issues:

- How can actions be initiated based on timed events?
- How can a state have internal states (composite state)?
- How can different state automata work side by side (orthogonal regions)?

Reacting to elapsed time is an important concept for state machines. Some examples might be:

- A traffic light control system that controls the respective duration of the red, yellow and green phases, either periodically, i.e. time-controlled or triggered by requests.
- In technical installations – from the simple coffee machine example through motor vehicles to nuclear power plants – maintenance intervals must be observed to ensure the proper functioning of the system. Maintenance might be triggered by specific performance issues of the system, or by elapsed time, whichever occurs first.
- An administrative act might become legally effective after the expiry of a period for its opposition, unless an objection is filed beforehand.
- In a data communication protocol with a positive acknowledgment method, a message may be deemed lost if the sender does not receive a reception acknowledgment from the recipient within a predetermined time (timeout).

All these scenarios can be modelled using state automata and either directly controlled by software (traffic light, data communication) or supported by it (maintenance, administrative act).

We will extend our blind control system by time-controlled events to show how to operate the blinds automatically, depending on the ambient brightness. In bright sun the blind should darken the room; if the brightness decreases, the blind should go up. We have already implemented a similar requirement: raising the blind in a strong wind to avoid damage. Since the wind speed always has priority over user requests, the modelling was quite simple, limited to two state transitions that trigger in strong winds, and an additional condition that prevents the user from moving the blind down.

Controlling the blind based on environmental brightness is more complex, because different influencing variables intermesh. On one hand the blind control should react to the ambient brightness as described and adjust the blind automatically. On the other hand, users' wishes should be taken into account: if someone wants bright sunlight in the room and operates a blind that has been lowered automatically, the blind control system should respect this choice and not try to oppose the user's actions. If the user does not operate the blind for an extended period of time – this is where the time aspect comes into play – automatic behaviour should be resumed.

The basic idea is simple and can be explained by the following state machine:

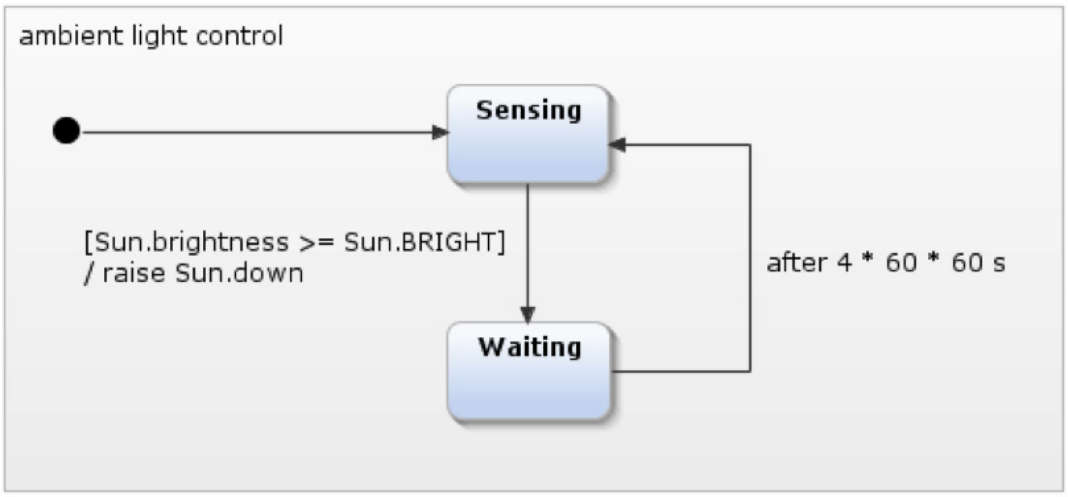

Fig. 5: Model of a brightness sensor with time delay

The statechart in Fig. 5 is limited to measuring the ambient brightness and initiating a corresponding response. Initially the state machine is in the state **Sensing**, in which it continuously polls the ambient brightness. As soon as the brightness – contained in the variable *Sun.brightness* – reaches or exceeds the specified *Sun.BRIGHT* threshold value, the machine performs the transition **Sensing** → **Waiting**. This transition triggers the *Sun.down* event, which is intended to shut the blind.

In the **Waiting** state nothing happens. Users can adjust the blind as they like and the automatic operation will not intervene as long as **Waiting** is active. However, this is not always the case: the transition **Waiting** → **Sensing** returns to the state in which the control system measures the sun's brightness and responds accordingly. This transition is triggered „after 4 * 60 * 60 s", i.e. four hours after activation of the **Waiting** state.

04.3 Combining several state machines

It would be nice if we could combine this simple state machine with the one we have already developed, so that the generated event *Sun.down* is effective there. The result looks like this:

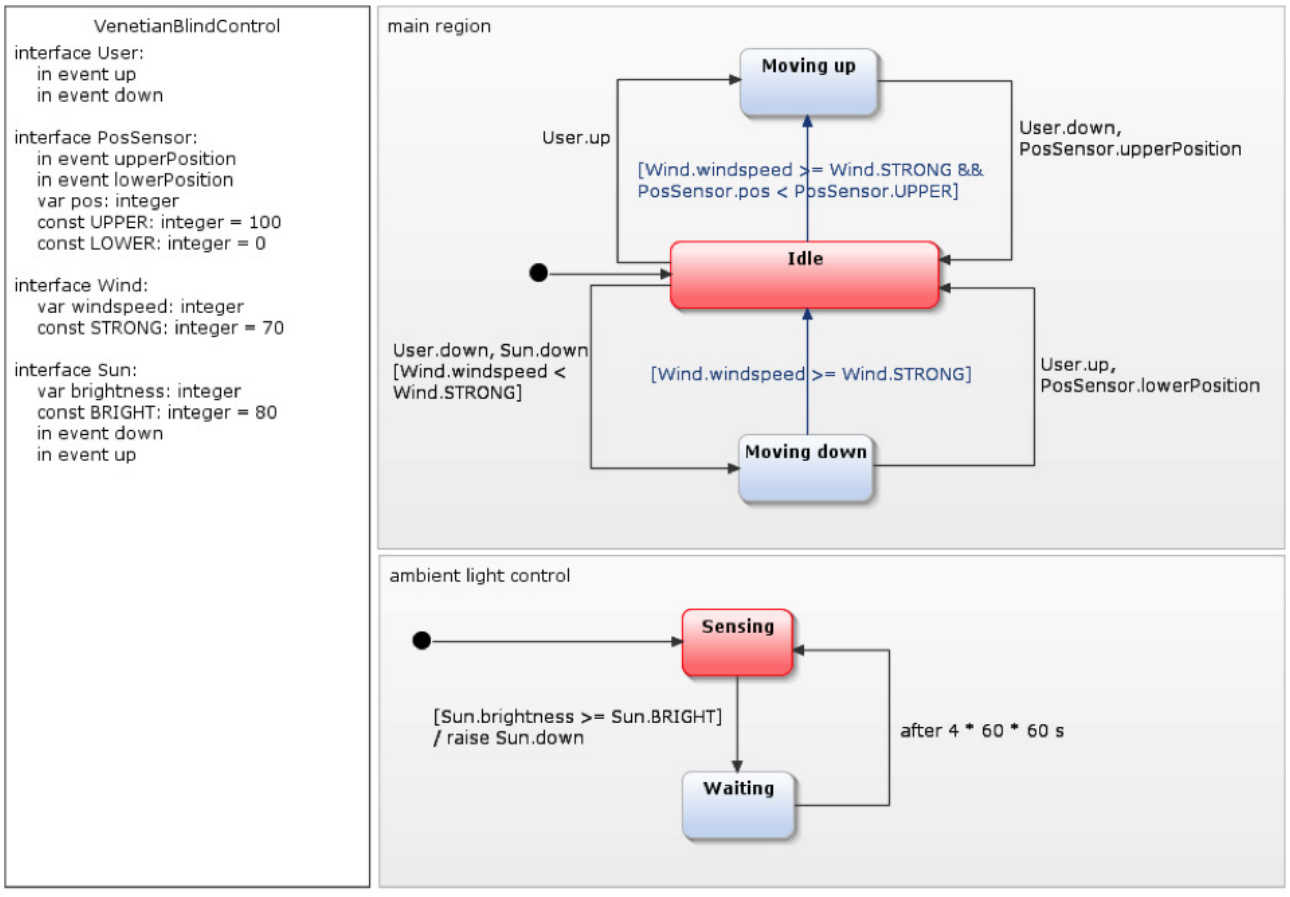

Fig. 6: Statechart with two orthogonal regions. The **Idle** and **Sensing** states are active at the same time, shown here in the simulation mode of YAKINDU Statechart Tools.

Fig. 6 shows two so-called *orthogonal* regions. Both contain their own state diagram. While the *main region* is waiting for a user request or strong wind, the *ambient light control* region simultaneously checks the ambient brightness. The state of the overall system thus consists of the states of its orthogonal regions. However, orthogonal ranges are not completely independent: when *ambient light control* changes from **Sensing** to

Waiting the operator *raise* generates a *Sun.down* event to which the *main region* should react and close the blind. For this to happen, the transition **Idle → Moving down** requires the *Sun.down* event as another triggering event.

The simulation mode of YAKINDU Statechart Tools allows us to enter the ambient brightness value. If the value of *Sun.brightness* is 80 or higher, the result looks like this:

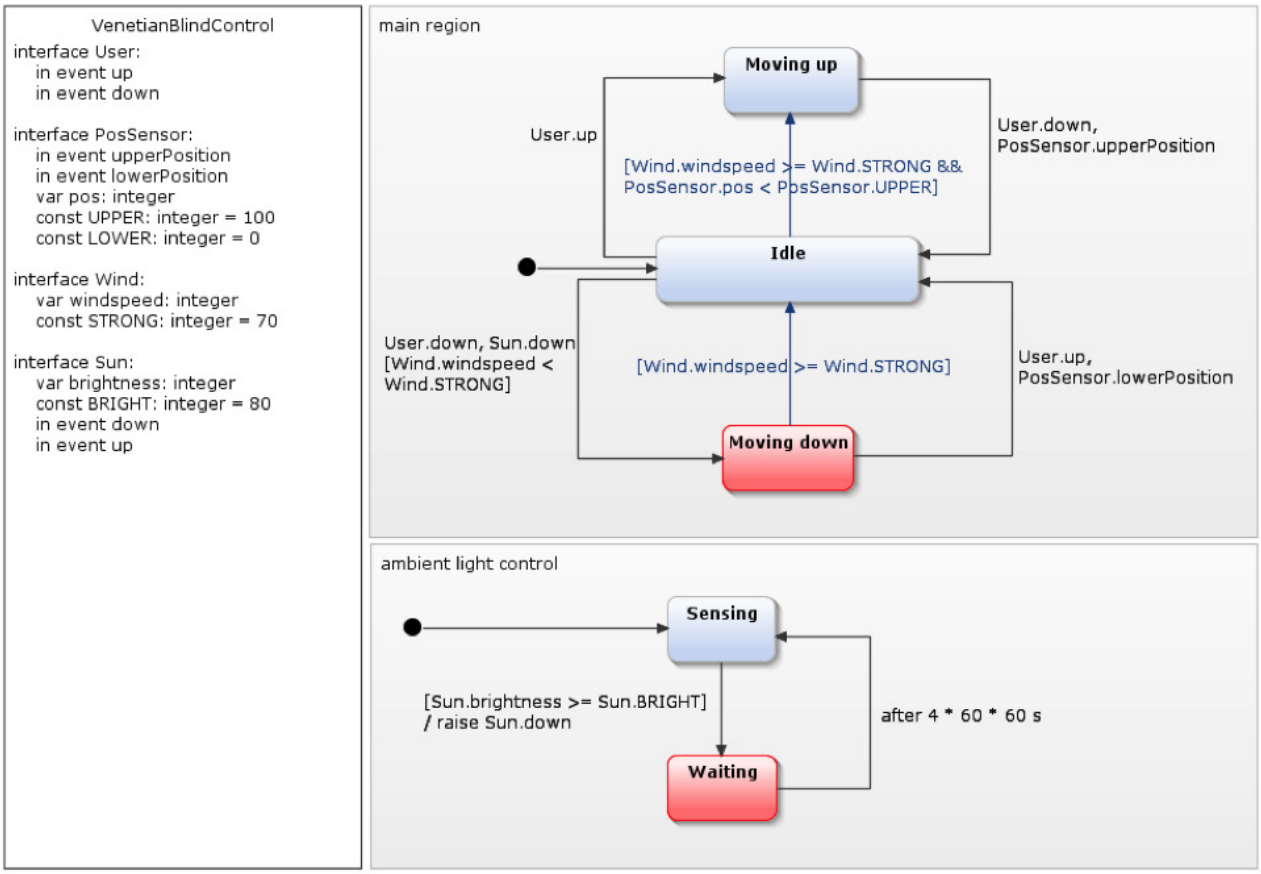

Fig. 7: Simulation of the blind control system at high ambient brightness

The blinds should not only close depending on the ambient brightness, but also open. If we want to support independent wait times for both functions, we duplicate the *ambient light control* area, adjust it for a brightness value, and extend the main region with a reaction to *Sun.up* in the same way as we did before:

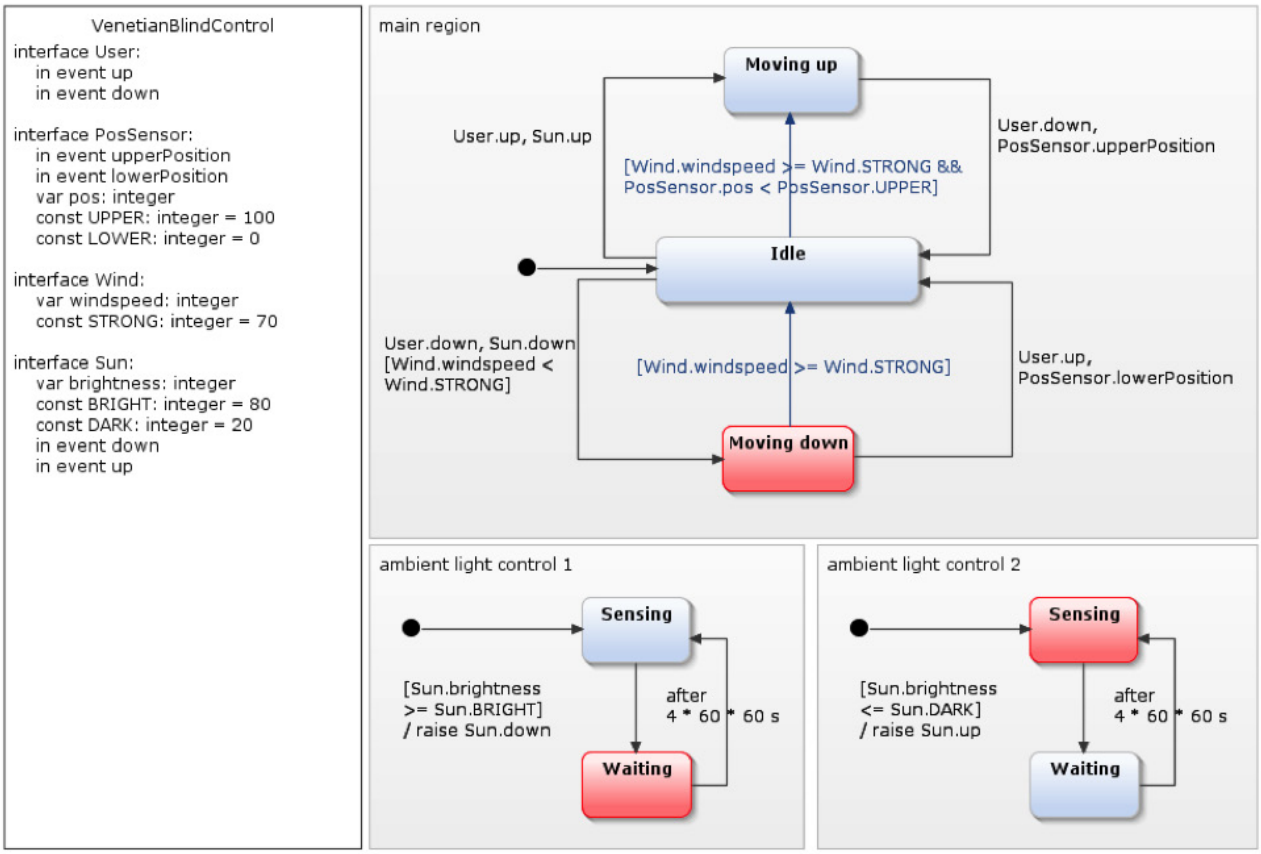

Fig. 8: The blind control system now reacts to overshooting and undershooting ambient brightness values

YAKINDU Statechart Tools executes orthogonal regions sequentially within repetitive processing cycles. This is relevant if the *raise* operator creates an event in region *A* that is to be processed in region *B*. However, each event only exists within the current processing cycle, so the execution of region *A* must occur before region *B* is reached. For more information on this, see section „Raising and processing an event" [5] in the YAKINDU Statechart Tools documentation.

04.4 Composite states

Now consider the following scenario: on a cloudy day the sun is continually breaking through, but soon disappearing behind the clouds again. In this case the blind should not be moved up and down continuously, nor should it close in response to the first bright but short sunny interval and then remain down for hours. It should restrict the sunlight only when it remains permanently bright outside, i.e. if the ambient brightness has not merely exceeded the threshold for a short time.

This behaviour is modelled as shown in Fig. 8. If the ambient brightness is below the threshold, the **Not bright** state is active. If the brightness exceeds the threshold value, the state machine changes to the **Almost bright** state. Effectively this is saying: "It is bright outside, but we are not sure if it will last long." We should therefore wait a little longer before we shut down the blind. As long as we are in the **Almost bright** state and the ambient brightness falls below the threshold, typically because a cloud blocks the sun, the state machine immediately goes back to **Not bright**,

possibly only to jump back to **Almost bright** shortly afterwards. The transition leading to **Waiting** is triggered and the blind is lowered only when **Almost bright** is active for 10 minutes without interruption. This works because the period specified as the operand of **after** is restarted each time the source state of the transition becomes active.

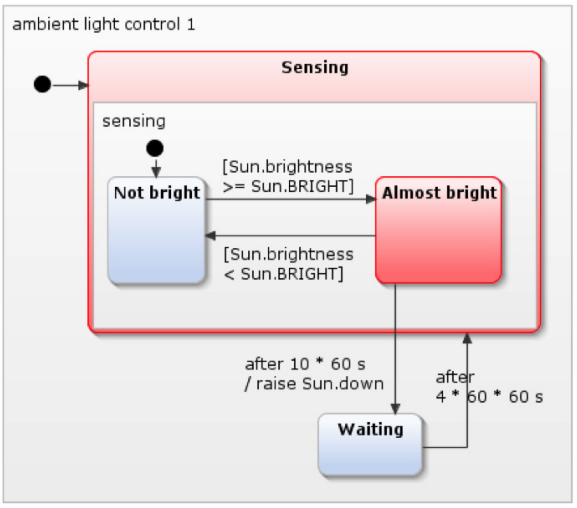

Fig. 9: Ignoring short-term brightness fluctuations

Fig. 9 also shows an extremely useful way to keep statecharts manageable and structured: the composite state. The more states and transitions an automaton contains, the less clear the graphical representation becomes, when it should actually provide clarity. It becomes difficult to keep a view of the whole machine and to recognize structures and logical connections. In the extreme, in state diagrams with hundreds or thousands of states, this becomes completely impossible.

Composite states put structure and modularization into the diagram. A composite state contains inner states and transitions and can thus encapsulate functionality. In the example of the blind control system the functionality of delayed darkening is in the composite state **Sensing**. A transition can lead to a compound state as such, but also directly to one of its inner states. The transition **Waiting → Sensing** shows an example of the former. Leaving a composite state can be done by transitioning from an internal state to a state outside the composite state, for example **Almost bright → Waiting**.

But a composite state as such can also serve as the starting point of a transition. Such a transition is triggered if the execution of the composite state reaches a final state or an exit node.

The composite state **Sensing** has only a single region. However, composite states may well comprise several orthogonal regions, as we have already seen in the top level of the model.

YAKINDU Statechart Tools allow the user to simplify the presentation further by hiding the interiors of a composite state in a subdiagram, as shown in Figure 10.

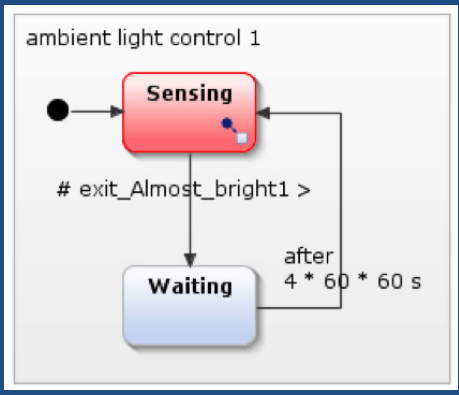

Fig. 10: Composite state with subdiagram

Sensing now looks almost like a „normal" state: only the small symbol at the bottom right of the state indicates that it is a composite state. This type of representation saves a lot of space. The chapter „Using subdiagrams" [6] in the YAKINDU Statechart Tools documentation discusses this further.

The graphical modelling language of YAKINDU Statechart Tools allows you to enter a composite state via any number of named entry points (entries) and exit via any number of named exit points (exits). For the sake of clarity and intelligibility the number of these entry and exit points, as well as the number of transitions in and out, should be kept as small as possible.

04.5 Further possibilities for improving the model

The response of the blind control system to changes in ambient brightness can be further improved. Instead of waiting for an extended period of time after each automatic opening or closing regardless of whether someone wants to change the blind's position, it would be more flexible and comfortable to only let the automatic operation wait for a specific time if the user actually presses the [↑] or [↓] button.

While we are at it we might also consider combining the two orthogonal regions *ambient light control 1* and *ambient light control 2* into a single region and thus simplify the machine. This is left as an exercise to the reader.

05. Implementation strategies for state machines

We should now have a better understanding of state machines, of the graphical modelling language's basic elements and of time-controlled transitions.

In the following sections we will examine how state machines models are turned into program code. Whoever models a blind control system and not only needs it as an example, will want to use it to create a real control unit, and so will need executable code. We will look at different implementation approaches for state machines and preview the automatic code generation abilities of YAKINDU Statechart Tools.

05.1 The large switch statement

Various approaches are possible for implementing a state machine in software. The most common one is using a switch statement. When using switch statements, developers often implement state machines, without being aware of it. Each state of the machine corresponds to one of the case clauses of the switch statement. The program jumps to the case clause that corresponds to the active state. Within this case clause, it executes state-specific statements, checks whether or which events are present, and possibly activates a different state depending on them.

In principle two variants are standard:

- In the event-driven approach, the state machine executes the switch instruction as soon as an event arrives.
- In the cycle-based approach, this is done at regular intervals.

A combination of both approaches is also possible. Executing the switch statement and processing the current events is called *run-to-completion step (RTC)*.

The basic structure of the implementation can be sketched very simply using the example of a traffic light, because each state of a traffic light is always assigned exactly one following state: red is followed by red-yellow, then green, then yellow and then red again. This state machine does – for now – not need any events, which results in a very simple and clear source code.

We choose Java as implementation language here, but of course this principle can also be implemented analogously in other programming languages. First we define an enumeration containing all states of the state automaton, i.e. the traffic light phases.

```
enum State {
     RED, RED_YELLOW, GREEN, YELLOW
}
```

The variable *activeState* contains the current status; it is initialized to red.

```
State activeState = State.RED;
```

The *stateMachine* method implements the actual state machine. Its core is a switch statement, which, depending on the current state, activates the next state by assigning it to *activeState*. In this example this occurs cyclically in an endless loop.

```
public void stateMachine() {
     while (true) {
          switch (activeState) {
          case RED: {
               activeState = State.RED_YELLOW;
               break;
          }
          case RED_YELLOW: {
               activeState = State.GREEN;
               break;
          }
          case GREEN: {
               activeState = State.YELLOW;
               break;
          }
          case YELLOW: {
               activeState = State.RED;
               break;
          }
          }
     }
}
```

The next example goes one step further and takes events into account. For this purpose we will return to the blind control system, as shown in the figure below.

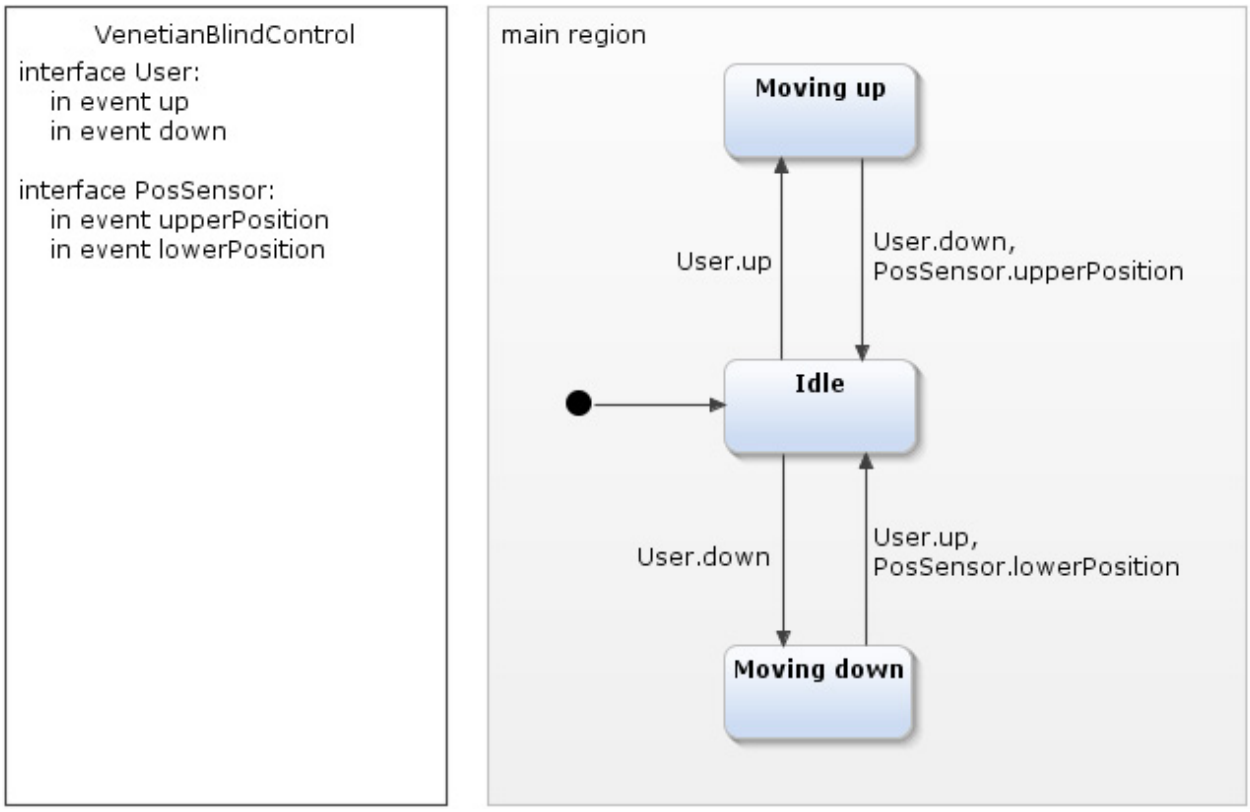

Fig. 11: Model of a blind control system: position sensor and events

The implementation in Java contains the states of the machine in the enum *State*:

```
enum State {
     INITIAL, IDLE, MOVING_UP, MOVING_DOWN
}
```

The events can also be represented in an enum:

```
enum Event {
     USER_UP, USER_DOWN,
     POSSENSOR_UPPER_POSITION, POSSENSOR_LOWER_POSITION
}
```

The variable *activeState* is preset with the initial state of the machine:

```
State activeState = State.INITIAL;
```

The *Collection<Event> events* records the incoming events:

```
Collection<Event> events = new ArrayList<Event>();
```

For developers who are unfamiliar with Java, *Collection<Event>* is a data type that manages a group of *Event* objects. The example uses an *ArrayList<Event>* as a concrete implementation, a list of *Event* objects that is implemented internally using an array.

The following idea lies behind this. As we have seen, the state machine continuously executes the switch statement in an endless loop. However, the state machine should be working only if there is actually something to do. This is motivated for example by the need to save power, or to prevent a multitasking system from unnecessarily hogging CPU capacity. The endless loop therefore does not run at the highest possible speed, but at a suitable, lower rate. That is, the machine executes one cycle (run-to-completion step) and then pauses for a while. This power saving effect is particularly important for embedded devices without external power supplies. In this example the waiting time between two cycles is 100 milliseconds:

```
long clockPulse = 100;
```

Events can arrive within this waiting period: how this happens we leave outside this discussion. In any case, these events enter the collection defined above and can be evaluated in the next processing cycle. If the blind control is in the **Idle** state and the *User.up* event is present, the state machine switches to the **Moving up** state. The event *User.down* triggers a

transition to **Moving down**. The Java code that implements this function looks like this:

```
case IDLE: {
     if (events.contains(Event.USER_UP))
          activeState = State.MOVING_UP;
     else if (events.contains(Event.USER_DOWN))
          activeState = State.MOVING_DOWN;
     break;
}
```

If any other events arrive while **Idle** is the active state they are ignored: a state machine should respond by definition only to those events the respective active state is „sensitive“ for.

Several events may be present simultaneously. The lower the clock rate, the greater the probability is that a user will press both the [↑] and [↓] keys between two processing cycles. In this case, in the above implementation the [↑] key always „wins“. If you want to give priority to the [↓] key, you must adapt the implementation accordingly.

Here is the complete method *stateMachine*, which implements the behaviour of the state machine:

```
public void stateMachine() {
    while (true) {

        /* Act upon the active state: */
        switch (activeState) {
        case INITIAL: {
            activeState = State.IDLE;
            break;
        }
        case IDLE: {
            if (events.contains(Event.USER_UP))
                activeState = State.MOVING_UP;
            else if (events.contains(Event.USER_DOWN))
                activeState = State.MOVING_DOWN;
            break;
        }
        case MOVING_UP: {
            if (events.contains(Event.POSSENSOR_UPPER_POSITION) ||
                events.contains(Event.USER_DOWN))
                activeState = State.IDLE;
            break;
        }
        case MOVING_DOWN: {
            if (events.contains(Event.POSSENSOR_LOWER_POSITION) ||
                events.contains(Event.USER_UP))
                activeState = State.IDLE;
            break;
        }
        default: {
            /* Should never happen. */
            throw (new IllegalStateException());
        }
        }
```

We have already seen that at most one event leads to a transition within a single clock cycle. This will have already happened at this point if there was a suitable event. Whether we are in a new state now or still in the same state as before, the events that arrived during the last cycle have no further relevance. For the next cycle we must clear them:

```
    /* Clear events: */
    events.clear();
```

The current processing cycle is thus completed. The state machine now pauses *clockPulse* milliseconds:

```
    /* Pause until the next run to completion step is due: */
    try {
        Thread.sleep(clockPulse);
    } catch (InterruptedException e) {
        // Ignore
        }
    }
}
```

05.2 The state machine as a table

In the automata theory, state transition tables [7] are often used to specify state machines. The source and target states, as well as the events that lead to a transition from the source to a target state, are listed in a table that the implementation refers to repeatedly.

5.2.1 One-dimensional state transition tables

A one-dimensional state transition table defines one transition per line. In the simplest case the table contains two columns: one contains all the source states, the other one the respective target states.
For a traffic light control it looks like this:

Active state	Next state
Red	Red-Yellow
Red-Yellow	Green
Green	Yellow
Yellow	Red

The reality is, of course, more complex: a traffic light should not be rushing through its states in a fraction of a second. Instead it should remain in the current state for a specific time, then change to the next state.

This behaviour can be approximated by a slow clock of, say, ten seconds. The state machine considers the table at every clock pulse, checks what the current state is and changes to the following state.

However, the result would still be very unsatisfactory, because the yellow phases would be much too long and the red and green phases would be much too short. We need different durations in the individual phases. This

can be achieved with time-controlled events, as already described. The change from the state **Red** to **Red-Yellow**, for example, takes place only after the occurrence of the event *20 s passed*.

In one-dimensional state transition tables each event is usually assigned its own table column. Transitions (= table rows) that should only take place when a certain event occurred contain a checkmark in the corresponding cell:

Current state	20 s passed	3 s passed	30 s passed	next state
Red	✓			Red-Yellow
Red-Yellow		✓		Green
Green			✓	Yellow
Yellow		✓		Red

The first line describes the state transition from **Red** to **Red-Yellow**. Since a check mark has been placed in the event column *20 s passed*, the transition takes place only after this event has occurred. The events *3 s passed* and *30 s passed* are irrelevant here, so the check mark is omitted in the respective columns.

05.2.1.1 Controlling a blind with a status transition table

The state transition table for the blind control system looks like this:

Current state	User.up	User.down	PosSensor. upper Position	PosSensor. lower Position	next state
Initial					Idle
Idle	✓				Moving up
Idle		✓			Moving down
Moving up		✓			Idle
Moving up			✓		Idle
Moving down	✓				Idle
Moving down				✓	Idle

05.2.1.2 The state transition table in the source code

This table can easily be converted into the source code of the desired programming language. We've already seen examples in Java, so here's something for C programmers:

```
#include <stdbool.h>
#include <stdio.h>

enum states {
    INITIAL, IDLE, MOVING_UP, MOVING_DOWN
};

enum columns {
    SOURCE_STATE,
    USER_UP, USER_DOWN, POSSENSOR_UPPER_POSITION, POSSENSOR_LOWER_POSITION,
    TARGET_STATE
};

#define ROWS 7
#define COLS 6
int state_table[ROWS][COLS] = {
/*        source,      up,    down,  upper, lower, target */
        { INITIAL,     false, false, false, false, IDLE },
        { IDLE,        true,  false, false, false, MOVING_UP },
        { IDLE,        false, true,  false, false, MOVING_DOWN },
        { MOVING_UP,   false, true,  false, false, IDLE },
        { MOVING_UP,   false, false, true,  false, IDLE },
        { MOVING_DOWN, true,  false, false, false, IDLE },
        { MOVING_DOWN, false, false, false, true,  IDLE }
};
```

The enum *states* contains symbolic names of the states, simplifying the creation of the table *state_table*. The enum *columns* assigns symbolic names to the column numbers. For example, the program below does not have to address the columns containing the initial and target states with their absolute indexes 0 and 5, but instead it uses the symbolic names *SOURCE_STATE* and *TARGET_STATE*. The symbolic names of the events are not needed in the sample program, but they are helpful for manual maintenance of the state transition table.

The table *state_table* specifies the state machine in a pure declarative fassion. If the machine changes, only adjustments to the table are necessary. This is manageable as long as the number of states and events is small. If that number should increase, however, clarity and maintainability fall by the wayside. After all, the actual program code remains unchanged – an advantage because changes are simpler and less error-prone.

We have already looked at the first part of the sample implementation in C. The function *main* is shown below. The program asks for the number of the next event, which the user enters via standard input. The event numbers correspond to the position of the events in the variable *columns*, i.e. USER_UP = 1, USER_DOWN = 2 and so on. The machine then consults the table. If it finds a transition that triggers the entered event, the program informs the user of the row and column in the table and returns the number of the new state. The state number correspond to the respective position in the enumeration *states*.

```
int main(int argc, char** argv) {
    int activeState = INITIAL;
    int event = -1;
    while (true) {
        int row, col;
        bool checkRows = true, checkCols;
        printf("Active state: %d\n", activeState);
        printf("Event: %d\n", event);

        /* Look for a row with the active state in the SOURCE_STATE column: */
        for (row = 0; checkRows && row < ROWS; row++) {
            checkCols = true;
            if (activeState == state_table[row][SOURCE_STATE]) {
                /* Found a row matching the active state. */
                /* Now check the columns for events triggering this transition: */
                for (col = SOURCE_STATE + 1; checkCols && col < TARGET_STATE; col++) {
                    if (state_table[row][col]) {
                        /* Found one. The current event must match this column
                         * to trigger the transition. */
                        if (event == col) {
                            /* The event matches. Let‘s do the transition: */
                          printf("Active state and event matching row %d, column %d.\n", row, col);
                            printf("Transitioning to target state %d\n", activeState);
                            activeState = state_table[row][TARGET_STATE];
                            /* Stop checking any further rows and columns: */
                            checkRows = checkCols = false;
                        } else {
                            /* The event does not match, so this transitions won‘t be triggered.
                             * We do not need to check any remaining columns: */
                            checkCols = false;
                        }
                    }
                }
                if (checkCols) {
                  /* At this point, we have checked all columns, but none of them requires an event
                     * for this transition. So we can do the transition in any case: */
                    printf("Transitioning without any event to target state %d\n", activeState);
                    activeState = state_table[row][TARGET_STATE];
                    checkRows = false;
                }
            }
        }
        printf("Active state: %d\n\n", activeState);
        printf("Please enter event number!\n");
        scanf("%d", &event);
    }
}
```

Attentive readers may notice that this implementation does not use a fixed clock, but is purely event-driven.

05.2.1.3 Conditions and actions

Unlike this example, transitions usually depend not only on events, but also on additional conditions (guard conditions). This means that for a transition to trigger not only the right event must be present, but also the guard condition must be fulfilled. In an implementation, a transition is associated with an evaluation function that checks the condition.

A state transition table that respects guard conditions does not encode just boolean values, but instead holds pointers to the relevant evaluation functions. In object-oriented languages this could be an object containing the evaluation function. If a table cell's value is *null*, this is equivalent to *false*, that is, there is no transition for this combination of state and event. However, if the table cell contains a *non-null* value, this value is regarded as the pointer to an evaluation function. First of all, this is equivalent to *true*; the transition is thus a candidate for being taken in principle. The implementation now calls the function. This checks the guard condition and, based on its return value, determines whether or not the automaton should actually perform the transition or not.

Transitions and states can be linked to actions. Considering them can also be left to the functions to which the state transition table refers. Such a function therefore not only checks the guard condition, but also activates the target state if the condition is fulfilled.

Concepts such as orthogonality, compound states or history states are difficult to implement using a table. In theory there is an equivalent representation for each of these constructs as a flat state machine, but in practice the state space explodes very quickly. For example, to map orthogonal

regions to a flat state machine there must be a separate state for each permutation of the active states in the orthogonal regions. The number of these states is given by the cross-product of the states in the orthogonal regions. In two orthogonal regions, each with three states, nine flat states may still be tolerable, but ten orthogonal regions, each with eight states, would result in 8^{10} flat states. That is more than a billion.

05.2.2 Two-dimensional state transition tables

Two-dimensional transition tables are generally more compact than one-dimensional tables, because a single row is sufficient per output state. The respective target state is noted in the cell that is defined by the intersection of source state row and event column. In the following table the event *User.down* (third column) in the **Idle** state (second row, disregarding the header row) leads to the target state **Moving down**.

Current state	User.up	User.down	PosSensor.upper Position	PosSensor.lower Position	no event
Initial					Idle
Idle	Moving up	Moving down			
Moving up		Idle	Idle		
Moving down	Idle			Idle	

The implementation basically looks the same as for one-dimensional tables. A truth value is no longer sufficient for the table cells; instead they must hold the respective target state. If we want to consider guard conditions and actions associated with transitions, we need a pointer to a function that corresponds to the transition and implicitly contains the target state. Generally there may be different functions reaching the same target state.

05.3 The State pattern

An object-oriented implementation variant is the State software design pattern [8] as implemented by Spring Statemachine [9], Boost Meta Static Machine (MSM) [10] or the Qt State Machine Framework [11]. The State pattern belongs to the group of behavioural design patterns [12], and encapsulates the state-dependent behaviour of an object as seen from the outside. Each state is implemented as a separate class that defines the behaviour of the machine in that state. All state classes are derived from a common interface, so that all states can be treated uniformly.

The class diagram of the blind control system is as follows:

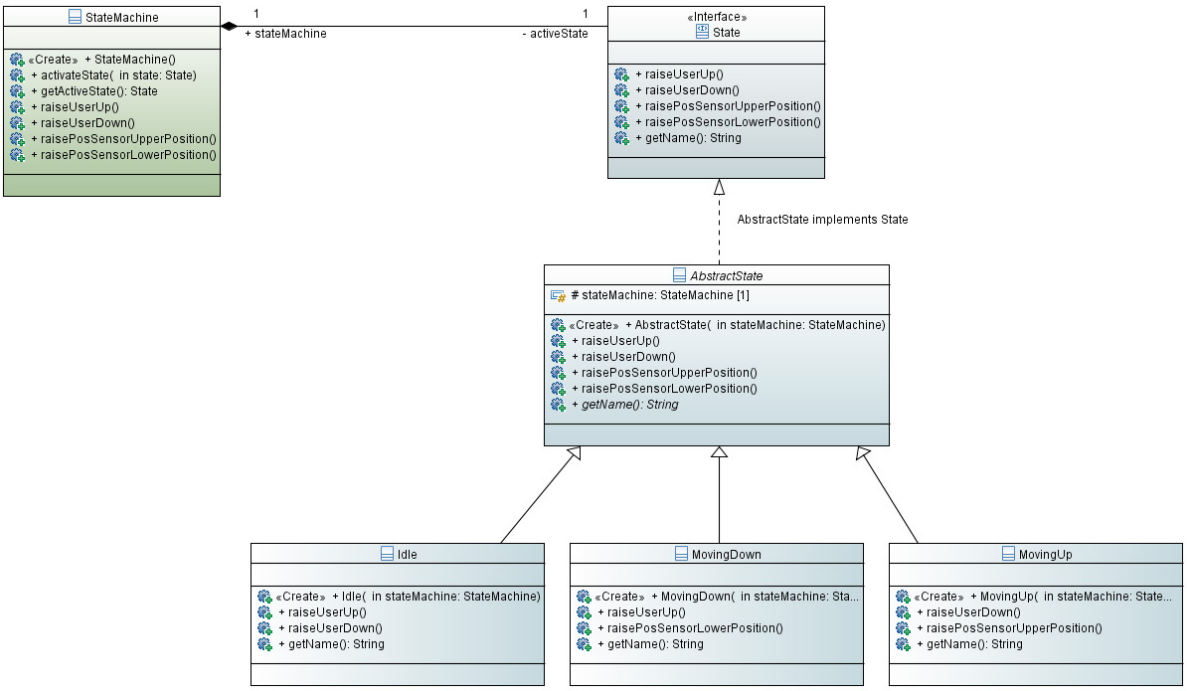

Fig. 12: Class diagram of the blind control system

The interface *State* defines the methods whose behaviour is state-dependent. In addition to the methods starting with *raise*... to trigger an event, there is the method *getName*, which returns the name of the state.

The *StateMachine* class represents the state machine. In its attribute *activeState* it manages the class corresponding to the active state and thus the state-specific behaviour. The State pattern is similar to the Strategy pattern [13]: a state corresponds to a strategy, i.e. the implementation of a concrete behaviour. However, while the strategy in the Strategy pattern is set from the outside, the states in the State pattern take care of this themselves. In the state pattern, if you like, a „strategy" replaces itself by another one – or in the correct nomenclature: a state activates its subsequent state on its own.

Let's take a closer look at the example. The client application sends messages to the state machine by calling *StateMachine* methods. Let's say the user presses the [↓] key. The client informs the state machine about this action by calling the *raiseUserDown* method. The response of the machine depends on the active state:

- In the **Moving up** state, the machine changes to the **Idle** state.
- In the **Idle** state, the machine switches to the **Moving down** state.
- In the **Moving down** state, the automaton ignores the event.

StateMachine therefore delegates the call to the corresponding method of the active state object. The *StateMachine* method *raiseUserDown* calls the same-named method of *activeState*:

```
public void raiseUserDown() {
    activeState.raiseUserDown();
}
```

What happens next is at the discretion of the state object. If it is an object of class *Idle*, a transition to **Moving down** occurs. This is what *raiseUserDown* looks like in *Idle*:

```
public void raiseUserDown() {
    stateMachine.activateState(new MovingDown(stateMachine));
}
```

The method activates the new state by passing a corresponding state object to the *StateMachine* method *activateState*, which assigns it to the *activeState* field.

If, on the other hand, the machine is in the **Moving down** state, nothing should be done when *raiseUserDown* is called. The *MovingDown* class therefore does not implement this method at all. Instead, the *raiseUserDown* implementation of superclass *AbstractState* is called:

```
public void raiseUserDown() {
}
```

This method does nothing and thus ignores the event – in our case, a meaningful default behaviour for all events for which the particular concrete state class does not implement any behaviour

05.3.1 Source code of the sample application

Let's look at the source code of the blind control system in an implementation based on the State pattern. The state machine consists of six classes, as well as a client class that uses the machine.

The interface *State* specifies what a state is in the sample application:

```
package de.itemis.state_machine.example.state_pattern;

public interface State {

    public void raiseUserUp();

    public void raiseUserDown();

    public void raisePosSensorUpperPosition();

    public void raisePosSensorLowerPosition();

    public String getName();

}
```

The *AbstractState* abstract class implements the reference to the *StateMachine* and the default behaviour for events, ignoring them:

```
package de.itemis.state_machine.example.state_pattern;

abstract public class AbstractState implements State {

    protected StateMachine stateMachine;

    public AbstractState(StateMachine stateMachine) {
        this.stateMachine = stateMachine;
    }

    public void raiseUserUp() {
    }

    public void raiseUserDown() {
    }

    public void raisePosSensorUpperPosition() {
    }

    public void raisePosSensorLowerPosition() {
    }

    abstract public String getName();

}
```

The classes *Idle*, *MovingUp* and *MovingDown* represent the three concrete states:

```
package de.itemis.state_machine.example.state_pattern;

public class Idle extends AbstractState {

    public Idle(StateMachine stateMachine) {
        super(stateMachine);
    }

    @Override
    public void raiseUserUp() {
        stateMachine.activateState(new MovingUp(stateMachine));
    }

    @Override
    public void raiseUserDown() {
        stateMachine.activateState(new MovingDown(stateMachine));
    }

    @Override
    public String getName() {
        return "Idle";
    }

}

package de.itemis.state_machine.example.state_pattern;

public class MovingUp extends AbstractState {

    public MovingUp(StateMachine stateMachine) {
        super(stateMachine);
    }

    @Override
    public void raiseUserDown() {
        stateMachine.activateState(new Idle(stateMachine));
    }
```

```
    @Override
    public void raisePosSensorUpperPosition() {
        stateMachine.activateState(new Idle(stateMachine));
    }

    @Override
    public String getName() {
        return "Moving up";
    }

}

package de.itemis.state_machine.example.state_pattern;

public class MovingDown extends AbstractState {

    public MovingDown(StateMachine stateMachine) {
        super(stateMachine);
    }

    @Override
    public void raiseUserUp() {
        stateMachine.activateState(new Idle(stateMachine));
    }

    @Override
    public void raisePosSensorLowerPosition() {
        stateMachine.activateState(new Idle(stateMachine));
    }

    @Override
    public String getName() {
        return "Moving down";
    }

}
```

The *StateMachine* class is the „face“ of the state machine that is shown to the outside:

```
package de.itemis.state_machine.example.state_pattern;

public class StateMachine {

    public StateMachine() {
        activateState(new Idle(this));
    }

    State activeState = null;

    public void activateState(State state) {
        activeState = state;
    }

    public State getActiveState() {
        return activeState;
    }

    public void raiseUserUp() {
        activeState.raiseUserUp();
    }

    public void raiseUserDown() {
        activeState.raiseUserDown();
    }

    public void raisePosSensorUpperPosition() {
        activeState.raisePosSensorUpperPosition();
    }

    public void raisePosSensorLowerPosition() {
        activeState.raisePosSensorLowerPosition();
    }

}
```

The state machine is used by the application code like this:

```
package de.itemis.state_machine.example.state_pattern;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class Example {

    public static void main(String[] args) throws IOException {
        StateMachine stateMachine = new StateMachine();
        stateMachine.activateState(new Idle(stateMachine));
        BufferedReader console = new BufferedReader(new InputStreamReader(System.in));
        while (true) {
            System.out.println("Active state: " + stateMachine.getActiveState().getName());
            System.out.print("Please enter event number! ");
            System.out.print("1: User.up, 2: User.down, 3: PosSensor.upperPosition, ");
            System.out.println("4: PosSensor.lowerPosition");
            String line = console.readLine();
            int eventNr = -1;
            try {
                eventNr = Integer.parseInt(line);
            } catch (NumberFormatException ex) {
                eventNr = -1;
            }
            switch (eventNr) {
            case 1:
                stateMachine.raiseUserUp();
                break;
            case 2:
                stateMachine.raiseUserDown();
                break;
            case 3:
                stateMachine.raisePosSensorUpperPosition();
                break;
            case 4:
                stateMachine.raisePosSensorLowerPosition();
                break;
            default:
                System.out.println("Unknown event number: " + eventNr + ".");
```

```
                break;
            }
        }
    }

}
```

By the way, an object-oriented programming language is not necessarily required for the State pattern. It can also be implemented with an imperative language such as C as long as the language supports function pointers. The core of the State pattern is that the active state reacts to events and activates another state, if needed.

In a C implementation each state is associated with a function (the „state function“) that implements the relevant actions. The state machine memorizes the active state by a pointer to its state function. This returns a pointer to the next state or more precisely: to the latter’s state function.

05.4 Which implementation approach is best?

For „real“ computers such as notebooks, desktops or servers the differences between implementations do not really matter. However, the situation is different for embedded devices, which are typically very limited in memory or CPU performance. The clear answer of which implementation is best is – it depends.

State machines are very fast. In a state machine with few states and few events, optimization considerations are hardly worthwhile. The situation is different with a large number of states or events. At this point, however, we won't ponder what exactly a "large" number is – 500?, 50,000?, 5 million? –, but we will look at a few basic considerations only.

A large number of states means that:

- The switch statement contains many cases.
- The (one-dimensional) state transition table becomes very tall.
- The state pattern requires many state classes.

If the number of events is large, this means that:

- Within the individual case clauses in the switch statement there is a lot to do, depending on the respective state.
- The state transition table becomes very wide.
- In the State pattern the individual state classes are quite extensive, depending on the respective state.

In all implementation scenarios, the execution time is composed of the following elements:

- Searching for the active state
- Searching for a transition based on the event(s)
- Activating the subsequent state

In addition, guard conditions and actions have their respective execution times. Since these are highly application-specific, however, we will not consider them further here. The activation of the subsequent state consists in each case of a simple assignment: we can ignore its time and memory requirements.

05.4.1 Execution time requirements

To find the active state, the switch statement performs at least one comparison, the situation in which the first case clause of the switch statement is the active state. At worst, the switch statement requires as many comparisons as there are states, i.e. if the last case clause corresponds to the active state. If all states are equally frequently active, the switch statement performs $z/2$ comparisons on average, where z is the number of states. The execution time is thus of the order of magnitude $O(z)$. This is followed by the analysis of whether specific events are present. This would trigger the corresponding transitions. The execution time for this is – with a similar consideration as for the states – of the order of magnitude $O(e)$, where e is the number of event types.

The execution time requirement for the lookup operation in a one-dimensional state transition table is also O(*z*) + O(*e*). The first term describes the row-by-row searching for the active state, the second term represents the search for a given event within the row. If there are several transitions from the active state to different subsequent states, then several rows usually have to be searched. However, this does not change the order of magnitude.

The object-oriented approach wins out here, with a time requirement that is independent of the number of states. Since the active state is already present as an object, the time requirement for „searching" the active state is of the order of magnitude O(1). Then there's the method call corresponding to the event. At first glance this may also look like O(1), but the effort is likely to be conservatively estimated at O(e), depending on the programming language and its runtime environment.

05.4.2 Memory requirements

Switch statement as well as state transition table must include all combinations of states and events, either in the program code of the switch statement or in the table data. The memory requirement is therefore of the order of magnitude O(*z* · e) in both cases. However, the table always occupies this memory space to its full extent, even if it is sparsely populated. The switch statement, on the other hand, only needs to encode those state/event combinations for which an event leads to a transition. On the other hand, a function pointer in the table occupies only the memory size of an address, which in general should be less than the code that a single state/event combination occupies on average.

In the State pattern each state class contains a method for each event that the respective state must take into account. Thus the order of magnitude is also $O(z \cdot e)$. As with the switch statement, however, only those event methods have to be coded that should affect something other than the general behaviour implemented in the abstract upper class. Only these methods require space for their program code.

05.4.3 ROM or RAM memory

It is important to consider the devices on which a state machine is to run. Object-oriented languages typically require a large runtime environment and more RAM. This can limit or prevent their use on embedded devices. The more program parts can be fitted into immutable ROM, the better. Executable code, as in the switch statement, falls into this category, as well as hard-coded state transition tables. In an object-oriented implementation of the State pattern you should reuse static state objects, rather than always dynamically creating new objects in RAM.

05.4.4 Debugging capability

It is also important to know how easily a statechart application can be debugged during development. A developer needs to be able to set breakpoints at specific points in the application to halt execution there and continue it step by step, to analyze why the application does not behave as expected. In a state class this is quite simple. Switch statements are also suitable. For tables, the developer can only set breakpoints in the functions to which the function pointers in the table refer. If these pointers are in the wrong cells, however, the analysis is tricky.

05.4.5 Clarity and maintainability

Generally the code of state machines is neither intuitive nor comprehensible. In this respect the State pattern is best, because it avoids a confusing and difficult-to-read switch statement. As long as a state does not have to handle too many different events, it remains manageable. New states require only new state classes; a change in the behaviour is limited to a change in the affected states. This reduces maintenance efforts.

In the case of automata with few states, on the other hand, the effort of an object-oriented implementation is not necessarily worthwhile, because a switch statement in this case would not be so confusing.

If a state machine framework is available and is suitable for the target platform from a resource requirements viewpoint, it should be used: comprehensibility and maintainability can only benefit.

06. Outlook – Generating source code automatically

All the implementation variants we have discussed, in principle suffer from a number of disadvantages:

- No form of implementation can come close to the state diagram in terms of clarity and intelligibility. The more extensive a state machine is, the more unreadable and unmanageable is its representation as program code or state transition table.
- The transformation of a state diagram into an implementation is complex and error-prone.
- Changes to the statechart mean changes to the implementation. This is again complicated, error-prone and in no way easy to maintain.

Implementation effort and complexity require tool support, so that a developer ideally only needs to take care of the graphical statechart. YAKINDU Statechart Tools provides code generation support for several progamming languages.

07. Summary

State machines are not merely theoretical constructs, but are of great use in the modelling of event-discrete systems. Our examples show various advantages of graphical modelling with state machines. The models are clear and easy to understand. Even complex contexts can be visualized and managed. Models developed as state machines are not only comprehensible to computer scientists, but also easy to understand for their clients. Mistakes and misunderstandings can be clarified quickly, or do not arise at all.

YAKINDU Statechart Tools provides extensive support for modelling state machines, including Harel statecharts. It also allows the simulation of state machines and automatic code generation.

08. List of sources

- [1] YAKINDU Statechart Tools, itemis AG, 2017, https://www.itemis.com/en/yakindu/statechart-tools/, viewed 2017-02-23

- [2] Finite-state machine, Wikipedia (English), 2017-04-12, https://en.wikipedia.org/wiki/Finite-state_machine, viewed 2017-04-20

- [3] YAKINDU Statechart Tools, Installation Guide, itemis AG, 2017, https://www.itemis.com/en/yakindu/statechart-tools/documentation/installation/, viewed 2017-02-23

- [4] YAKINDU Statechart Tools, Tutorials, itemis AG, 2017, https://www.itemis.com/en/yakindu/statechart-tools/documentation/tutorials/, viewed 2017-02-23

- [5] YAKINDU Statechart Tools, User Documentation, "Raising and processing an event", itemis AG, 2017, https://www.itemis.com/en/yakindu/statechart-tools/documentation/user-guide/#raising-and-processing-an-event, viewed 2017-02-23

- [6] YAKINDU Statechart Tools, User Documentation, "Using subdiagrams", itemis AG, 2017, https://www.itemis.com/en/yakindu/statechart-tools/documentation/user-guide/#using-subdiagrams, viewed 2017-02-23

- [7] State transition table, Wikipedia (English), 2016-10-24, https://en.wikipedia.org/wiki/State_transition_table, viewed 2017-02-23

- [8] State Pattern, Wikipedia (English), 2017-03-26, https://en.wikipedia.org/wiki/State_pattern, viewed 2017-04-20

- [9] Spring Statemachine, Pivotal Software, 2017, https://projects.spring.io/spring-statemachine/, viewed 2017-02-23

- [10] Meta State Machine, Christophe Henry, Boost C++ Libraries, 2016-12-27, http://www.boost.org/doc/libs/1_63_0/libs/msm/doc/HTML/index.html, viewed 2017-02-23

- [11] The State Machine Framework, The Qt Company, 2017, http://doc.qt.io/qt-5/statemachine-api.html, viewed 2017-02-23

- [12] Behavioral pattern, Wikipedia (English), 2017-01-02, https://en.wikipedia.org/wiki/Behavioral_pattern, viewed 2017-02-23

- [13] Strategy pattern, Wikipedia (English), 2017-02-07, https://en.wikipedia.org/wiki/Strategy_pattern, viewed 2017-02-23

09. The author

Rainer Klute, born in 1961, is a software architect at itemis AG with a focus on Java, Linux and documentation.