Specification of Embedded Systems
Summer Semester 2020

Session 3

Modelling Behaviour, Part I
Operations – Statemachines – Events – Timing Conditions

Jan Peleska
peleska@uni-bremen.de

Issue 2.1
2020-06-06

Chapter 1

Preface

In this document, the material for **Session 3** of the course **Specification of Embedded Systems** is provided.

This document is structured as follows. Overview

- In Section 2 it is explained how a block can be associated with different variants of behaviour.
- In Section 3, an introduction to modelling behaviour with SysML state machines is given. But beware! The syntactic variants for modelling states and transitions are numerous, and their interpretation is quite complex. Therefore, you need to study the literature referenced there as well.
- An example how to use opaque behaviour as the classifier behaviour of an active block is given in Section 4.
- In Section 5, design guidelines for modelling blocks whose classifier behaviour is specified by means of state machines.
- As usual, these lecture notes end with questions and exercises in Section 6.

Contents

List of Figures

Chapter 2

Associating Blocks With Behaviour

Variants of behaviour

You may recall from Session 2, Section 2.2, that behaviour can be modelled in SysML using activities, interactions, state machines and use cases. In addition to these, there is **opaque behaviour** consisting of a textual specification of behaviour in syntax which is *outside* the UML/SysML. Typically, opaque behaviour is specified in the syntax of some programming language, we will always use C for this purpose. When a code generator creates code from a model, the opaque behaviours are used "as is" and inserted into the generated code without further modification. A specific language subclass of opaque behaviour is **function behaviour** which contains opaque specifications about input-to-output transformations without referencing or modifying state data in the execution context. Therefore, function behaviour can be described as a function in the mathematical sense, mapping arguments to image values.

While blocks have been introduced as the central SysML language elements for modelling structure, they can be associated with behaviour in the following ways.

Owned behaviour vs. nested classifier

- Instances of each of the behavioural modelling variants can be associated as **owned behaviour** with a block. "Owned" means in this context that the behavioural models can refer to the block context (value properties, ports, operations, ...) when evaluating expressions or changing state.

  In the Papyrus tool, owned behaviour is created by the following steps.

1. In the model explorer, open the context menu of the block to be associated with a new behaviour.
2. Select one of the behaviour variants listed above in the context menu.
3. For each of these behaviour variants, it is possible to select As owned Behaviour, or, alternatively, As nestedClassifier.
4. Choose "As owned Behaviour" to create the selected behaviour variant, so that it becomes owned behaviour of the block.

- Blocks can be used as containers for behaviours that do *not* use the block as a context but only as a namespace. These behavioured classifiers are created as and denoted by **nested classifiers**. We will not apply nested classifiers in this course, their more detailed description is given in [3, 13.2.3.4].

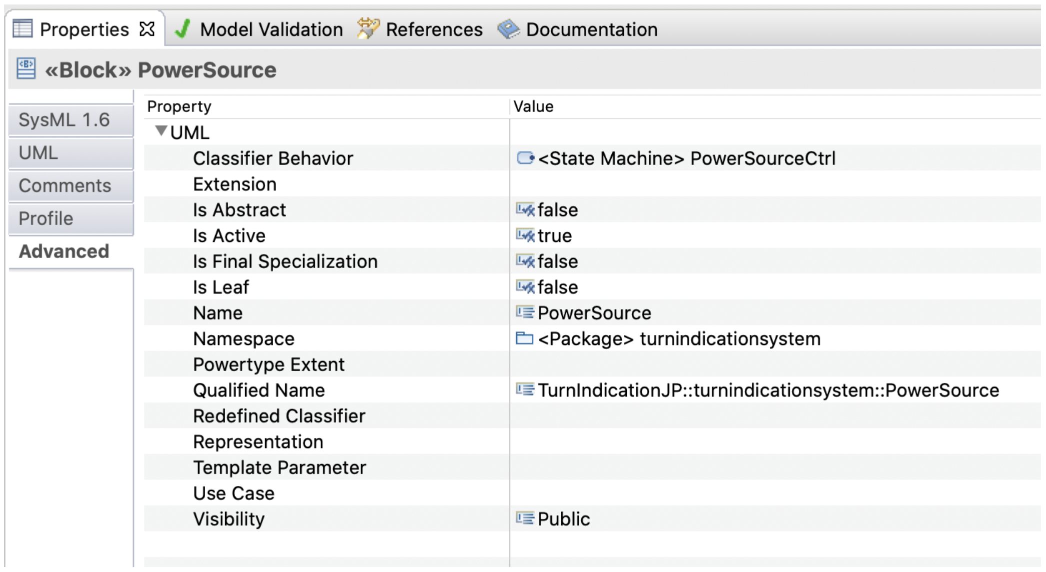
Active blocks and classifier behaviour

Blocks can be **active** in the sense that they can execute one specific owned behaviour as soon as an instance of the block is created. If the block represents software, this means that the instance starts to run as soon as it is instantiated as an independent thread or process. To model an active block, two modelling steps are necessary.

1. The block attribute Is Active must be set to `true`
2. One of the block's owned behaviours must be identified as the **classifier behaviour**: this is the behaviour which immediately becomes active when the block instance is created.

   In the Papyrus tool, the classifier behaviour is selected in the Advanced menu of the block by double clicking the Value entry which is initially empty. Then it is possible to select one of the block's owned behaviours as classifier behaviour. Note that this only works if the Is Active flag has been set.

Example 1. In the TurnIndication model for Session 3, you can find an active block `TurnIndicationJP::turnindicationsystem::PowerSource`. Its Advanced specification looks as shown in Fig. 2.1. Its classifier behaviour is modelled by state machine PowerSourceCtrl which has been introduced before as an owned behaviour of the block. Its Is Active flag is set to `true`. □



«Block» PowerSource

Property	Value
▼UML	
Classifier Behavior	<State Machine> PowerSourceCtrl
Extension	
Is Abstract	false
Is Active	true
Is Final Specialization	false
Is Leaf	false
Name	PowerSource
Namespace	<Package> turnindicationsystem
Powertype Extent	
Qualified Name	TurnIndicationJP::turnindicationsystem::PowerSource
Redefined Classifier	
Representation	
Template Parameter	
Use Case	
Visibility	Public

Figure 2.1: Advanced specification page for an active block with a state machine specifying the classifier behaviour.

The explanations given above and in Section 4 blow should suffice for the modelling activities in this course. If you would like to read more about blocks and associated behaviours, please consult [1, 7.5]. Further reading

Chapter 3

Specifying Behaviour With State Machines

3.1 Introductory Remarks on State Machines

Both in computer science and engineering, **state machines** are the preferred means for modelling so-called **reactive behaviour** in discrete control problems:

- The state machine resides in a given **state**[1] and waits for any input to occur.
- If an input occurs, the machine reacts by producing a (possibly empty) output and by transiting into the next control state (which may be a new state or the same as before).
- The input-output behaviour is determined by the control state the machine currently resides in.

If you are not familiar with state machines, please study the definition of a **Mealy Machine**, for example, in `https://en.wikipedia.org/wiki/Mealy_machine`. Otherwise, the concepts to be introduced next might be too complex to understand. The state machines used in UML and SysML are based on the **Statecharts** originally invented by David Harel, who has written a very precise article about their behavioural semantics [2]. I have

[1]often also called **control state**, to distinguish them from state information stored in value properties, ports and other attributes of a block

made this article available in Stud.IP, together with the other material for Session 3.

3.2 State Machines With Simple States

To start with the explanation of SysML state machines, consider the state machine LampCtrlSlave in Fig. 3.1, which is used to specify the classifier behaviour of block `TurnIndicationJP::turnindicationsystem::software::LampCtrlSlave`. The state machines reacts on CAN messages by setting the output port switchOut to `true` or `false`, respectively. As can be seen from the internal block diagram `TurnIndicationJP::turnindicationsystem::DoorController::DoorControllerIbd`, this output port is connected to a corresponding input port of block `TurnIndicationJP::turnindicationsystem::PowerSource`: value `true` transmitted from switchOut to switchIn has the effect that the PowerSource opens the 2mA power supply for the lamps. On switch value `false`, the power source is closed again.

Simple states

The states depicted as boxes with rounded corners are called **simple states**, as opposed to composite states to be discussed below. When entering such a state, an **entry action** is executed, if such an action has been defined. We specify entry actions as opaque behaviours in C-style syntax, reading from and writing to value properties or ports. In simple (and composite) states, time passes, until another transition becomes enabled.

Initial pseudo state

The initial state marked by a black bullet • is one of the **pseudo states** available SysML state machine syntax. Pseudo states are characterised by the fact that they must be left immediately, so that no time is spent while residing in a pseudostate. The transition arrow emanting from the initial pseudo state marks the first simple state to be entered when the state machine is executed.

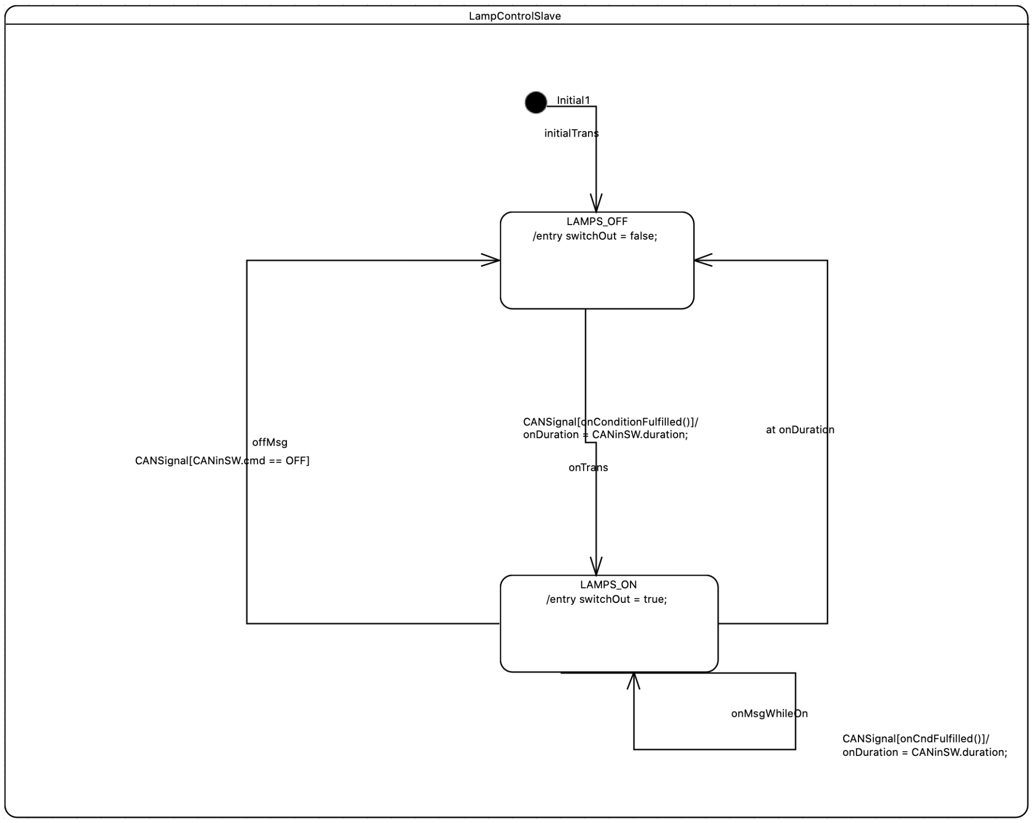

LampControlSlave
Initial1
initialTrans
LAMPS_OFF
/entry switchOut = false;
offMsg
CANSignal[CANinSW.cmd == OFF]
CANSignal[onConditionFulfilled()]/
onDuration = CANinSW.duration;
onTrans
at onDuration
LAMPS_ON
/entry switchOut = true;
onMsgWhileOn
CANSignal[onCndFulfilled()]/
onDuration = CANinSW.duration;

Figure 3.1: LampCtrlSlave: a state machine with simple states.

Transitions are labelled by

Transitions labels

1. An optional **trigger** which may represent one of the event types

   change-event, signal-event, time-event, call-event, any-receive-event

   We will use only the first three of these event types.

   Please read [3, 13.3] for an explanation of these events. Their basic intuition is

   - A **change-event** indicates that some constraint changed from `false` to `true`, because some of the value properties, ports, or other symbols referenced in the constraint changed their values.

- A **signal-event** indicates the occurrence of some event similar to an interrupt.
- A **time-event** indicates that a certain time has passed after the source state of the associated transition had been entered (after 200ms means "200ms after the source state had been entered), or that an absolute/date time event has been reached (at 2020-05-17, 11:45).

2. An optional **guard condition** which is a constraint over value properties, ports etc. in the scope of the block owning the state machine behaviour. We will always use **opaque expressions** to specify constraints.

3. An optional **action** to be executed if the transition **fires** (i.e. the transition is executed). We will always use opaque behaviour to specify the effect of such an action.

In Fig. 3.1, the transition from simple state LAMPS_OFF to LAMPS_ON is labelled as follows.

- The transition has a signal event as trigger, the associated signal is named CANSignal an we use it to indicate that a new CAN bus messages has arrived and can be read from the port CANinSW.[2]

- In the guard condition, Boolean block operation onConditionFulfilled() is called. You can inspect in the model how this operation is declared in block LampControlSlave, and you can inspect that its behaviour is again opaque (OnCndCheck) and has been specified in C-style by

```
return
(CANinSW.cmd == ON &&
((CANinSW.applyLeft && PinProgram == LEFT) ||
 (CANinSW.applyRight && PinProgram == RIGHT)));
```

  This means that the guard evaluates to true if and only if

  1. the CAN message command is ON, and
  2. the CAN message indicates that flashing should occur on the left-hand side and pin programming specifies that this lamp control slave handles a left-hand-side lamp, or

[2]For reasons unknown, the Papyrus tool does not show the signal *event* (which is CANMsgReceived in this case), but the signal name itself.

3. the CAN message indicates that flashing should occur on the right-hand side and pin programming specifies that this lamp control slave handles a right-hand-side lamp.

- As associated action, duration value of the CAN message is read from the port and written to a value property of the block, called onDuration.

A transition is **enabled** if

Enabling transitions

1. the state machine resides in the transition's source state[3],
2. the trigger has occurred, and
3. the guard condition evaluates to `true`.

If several transitions are simultaneously enabled, then only one of them fires, and this choice is nondeterministic.

Compound transitions

By using **fork** and **junction** pseudo states **compound transitions** can be created. For a valid SysML model, at least one guard condition emanating from a pseudo state must evaluate to `true`, since we must not get stuck in pseudo states: they shall be left in zero time. For this reason, transitions emanating from pseudo states may not possess any triggers, since it is not allowed to wait for event occurrences in a pseudo state.

Run-to-completion paradigm

Compound transitions and ordinary transitions are executed in zero time, and all transitions that may simultaneously fire (from concurrent state machines) must come to an end in a simple state, before the next input or event can be processed. Loops of compound transitions containing pseudo states only are forbidden, since this would model an infinite loop taken infinitely fast (a so-called **livelock**).

3.3 Composite States and Submachine State

States are called **composite** if they are decomposed hierarchically or into parallel **regions** containing sub-ordinate state machines. A **submachine state** `s` is a state which is not decomposed, but *references* a sub-state machine to be executed while the high-level state machine resides in `s`.

Further reading – manda-tory!

The combination of composite states with compound transitions results

[3]For transitions emanating from a composite state, this means that the machine resides in a simple state "underneath" this composite state.

in more complex firing rules. These lecture notes only give a short summary of what can be done with SysML state machines. You need to become acquainted with the details; therefore, **please read** [3, 14.1, 14.2] to get a comprehensive understating about available pseudo states, compound transitions, and compound states. In addition (and because it is easier to understand) read [1, Chapter 11].

3.4 Summary: Comparison Between SysML State Machines and Mealy Machines

Comparison SysML state machines vw. Mealy Machines

Summarising the most important extensions distinguishing SysML state machines from simple Mealy Machines are as follows.

- SysML state machines react on and produce different classes of events (signal events, change events, time events, call events), and events may carry data, whereas Mealy Machines only distinguish between atomic input events and likewise atomic output events.
- SysML state machine transitions not only depend on an event occurrence, but also on a guard condition.
- SysML state machines may change value properties and port states by writing to them in the form of assignments.
- SysML state machines distinguish between simple states, composite (hierarchic or parallel) states, submachine states, and pseudo states.
- SysML simple and composite states can have entry actions, exit actions, and do actions.

Chapter 4

Using Opaque Behaviour Specifications as Classifier Behaviours of Active Blocks

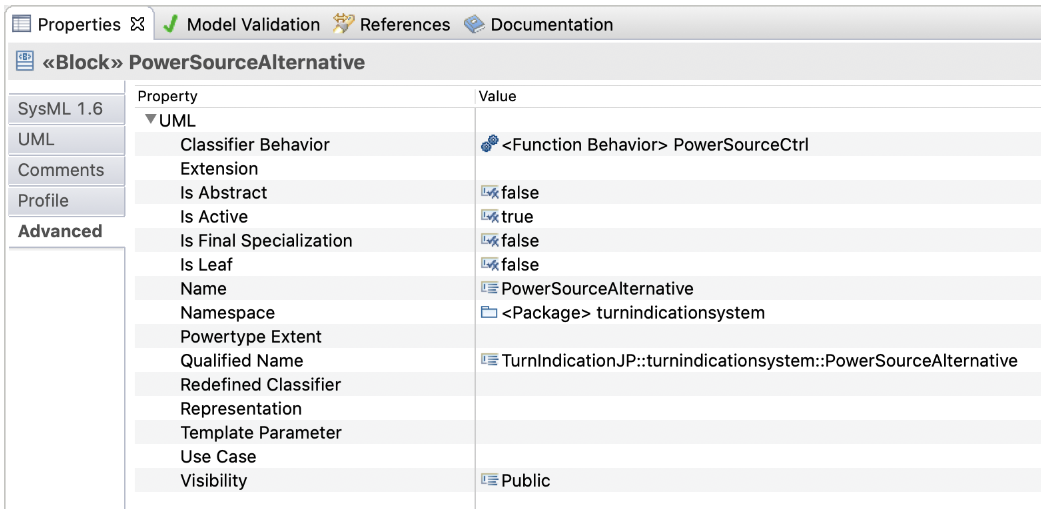
.

As an example where opaque behaviour is used to specified classifier behaviour, please inspect block PowerSourceAlternative in the TurnIndication model for Session 3: as owned behaviour, the (opaque) function behaviour PowerSourceControl has been specified with C-code

```
while (1) {
    currentOut = (switchIn) ? 2 : 0;
}
```

This is the body of a non-terminating thread running in the context of PowerSourceAlternative. In each loop cycle, the `currentOut` port is set to 2, if the `switchIn`-port carries value `true`. Otherwise, `currentOut` is set to 0.

In Fig. 4.1, the block's Advanced specification page is shown.



Properties | Model Validation | References | Documentation

«Block» PowerSourceAlternative

SysML 1.6 | UML | Comments | Profile | **Advanced**

Property	Value
▼UML	
Classifier Behavior	<Function Behavior> PowerSourceCtrl
Extension	
Is Abstract	false
Is Active	true
Is Final Specialization	false
Is Leaf	false
Name	PowerSourceAlternative
Namespace	<Package> turnindicationsystem
Powertype Extent	
Qualified Name	TurnIndicationJP::turnindicationsystem::PowerSourceAlternative
Redefined Classifier	
Representation	
Template Parameter	
Use Case	
Visibility	Public

Figure 4.1: Advanced specification page for an active block with function behaviour as classifier behaviour.

Chapter 5

Design Guidelines for Blocks and State Machines

5.1 Separate Blocks vs. Nesting of Blocks

Nested classifiers ...

You have noticed when working with the Papyrus tool that blocks can be created within blocks: a block (like an UML class) may have **nested classifiers**. The word 'classifier' is a general term for language elements describing different features. Block, ports, properties, signals are all specific variations of classifiers, and therefore, they are allowed to nest inside a block. The most important aspect of nesting is that the block "hosting" all these nested classifiers provides a **namespace** for them: the classifiers nested in a block have access to the block's attributes (which may all be nested classifiers), and so a state machine nesting in a block has access to its ports and value properties, just to name one example.

... vs parts

However, blocks nesting in blocks do *not* automatically become *parts* of the host block. You remember from the first sessions that an instance of a block becomes part of another block by specifying a directed (composite or shared) association between the block denoting "the whole thing" and the block whose instances are parts of the whole. As an example, please study again the TurnIndicationConttroller block and check how instances of front controller, door controllers, and rear controller became parts of the the TurnIndicationConttroller, as can be seen both in the project explorer and in the block definition diagram of the TurnIndicationConttroller.

As a guideline, do *not* nest blocks `B` inside another block `A`, if instances of `B` are intended to become parts of `A`. Instead, specify `B` separately and introduce the part relations later on in a BDD associated with `A`. This gives you also more freedom to decide where to "allocate" a part after your have designed its block. If the `B`-instances that become parts of `A` need information from `A`, this should be conveyed via ports and flows.

Design guideline

So when should you use nesting blocks? To answer this, it helps to understand that UML understands classes nesting in other classes as so-called **inner classes**, as we know them from Java or **nested classes** as they are called in C++. Inner classes are only used in cases where they require the complete context of their host classes and perform a specific service for the host class. For example, in a Java class representing a collection, its iterator is typically implemented as an inner class of the collection class.

Nesting blocks are like inner classes

As you can see, this example is focused on software. For HW/SW codesign, the context of the "outer box" (which is typically HW), as far as relevant to "inner boxes" (hardware parts and/or software parts should always be provided via ports and flows.

5.2 Block Configuration

When re-using a block owning a classifier behaviour by creating different instances as parts to be integrated in other blocks, it is often important to **configure** these parts differently, in order to adapt their behaviour to the specific part allocation. As an example, we have already seen that the LampControlSlave could be modelled as a re-usable block with classifier behaviour specified by a state machine. This block, however, is sometimes instantiated as a part controlling a left-hand side lamp, sometimes as a part controlling a right-hand side lamp.

Reuse requires configuration

As a guideline to model configurations I suggest to proceed in the following steps.

Modelling configurations

1. Collect all configuration parameters and specify them as value parameters (boolean flags, enumeration codes, integer codes, name strings ...) of an output port nested in a block `C` specialised on providing configuration data.

   In our turn indication controller project, we have created the PinProgramBlock for this purpose.[1] It is associated with a single port PinProgramOut which just conveys a single value of type LeverPosition (the concrete values are `LEFT` and `RIGHT`).

2. For every configuration needed, create a new block `Ci` which is a specialisation of `C`. This is best done on a block definition diagram for `C`, where a **generalisation** association is drawn from each `Ci` to `C`.

   In our turn indication controller project, this can be seen in BDD PinProgramBlockBdd: we need one configuration for flashing on the left-hand side and one for the right-hand side.

3. In the derived blocks `Ci`, specify a **constraint property**[2] as an opaque Boolean expression which specifies the value as required for the specific configuration.

   In BDD PinProgramBlockBdd, you can see how the derived block PinProgramBlockLeft has a constraint `PinProgramOut == LEFT`. This means that the port can only convey the constant value `LEFT`. Analogously, a constraint has been specified for derived block PinProgramBlockRight.

5.3 State Machine Initialisation

Cyber-physical control systems come with various challenges that may lead to considerable model complexity. One of these challenges is that every component in a complex control system may be restarted while the system is in operation. Think of an aircraft flying from Hamburg to Chicago: it occurs quite often that one of the many aircraft controllers (there are more than 200 in modern aircrafts) needs to be re-booted during flight. As a consequence, when modelling controller behaviour, it cannot be expected that a component start only happens when the whole system is started and all interfaces have default values.

[1]The term **pin programming** comes from the old days where computers or computer boards were configured using mechanical switches on the board to set some processor pins to constant zero or constant one. The value of these pins determined the behaviour of the computer board.

[2]This is done by opening the context menu of the block in the model explorer and selecting {?} Constraint from the New Child sub-menu.

Therefore, when modelling state machines, it is advisable to specify a collection of choice pseudo states and associated transitions that decide in which simple state the state machine should commence. Typically, the guard conditions involved check the current value of each interface that may influence the state machine behaviour. As an example, consider state machine ControlLogic in Fig. 5.1.

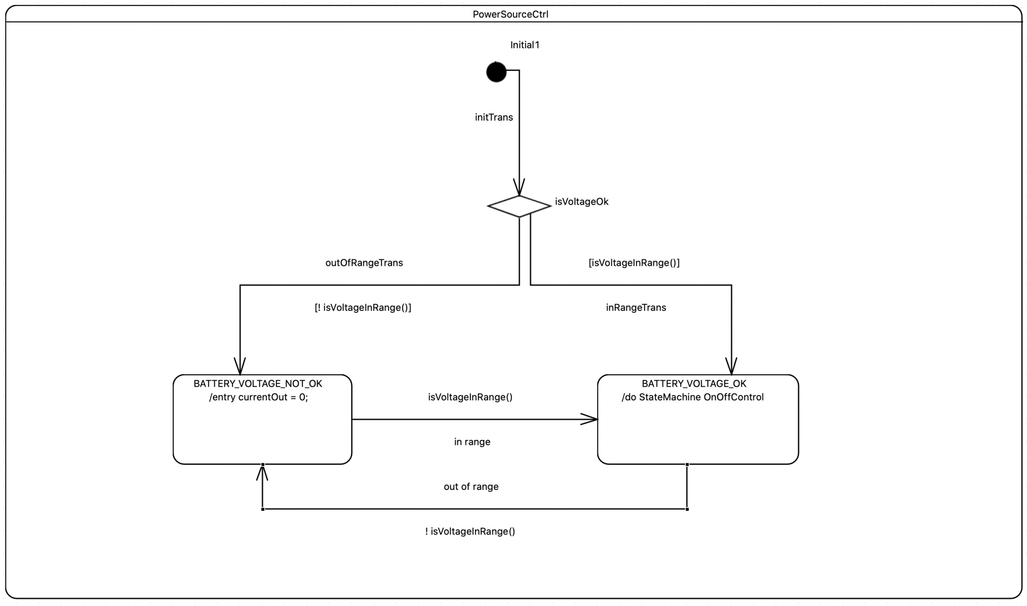


Figure 5.1: State machine PowerSourceCtrl, allocated in each of the controllers.

When being initialised, the Boolean operation `isVoltageInRange` is evaluated in choice pseudo state isVoltageOk. This operation checks whether the battery voltage is in admissible range, it has an opaque behaviour specified by

```
    return (BatVolIN.voltage >= 10 &&
            BatVolIN.voltage <= 15);
```

This means that the operation returns `true` if and only if the voltage is greater or equal than 10V and less or equal than 15V.

If the voltage is out of range, simple state BATTERY_VOLTAGE_NOT_OK is entered, and the lamp current is switched off, regardless of any switch commands coming from the LampControlSlave. If the voltage is ok, however, the hierarchic machine state BATTERY_VOLTAGE_OK is entered, where lamp current is switched off and on according to the commands of LampControl-Slave.

Don't rely on change events

It is usually a design error to enter always the same initial state (for example, BATTERY_VOLTAGE_NOT_OK) and then rely on a change event occurring and transferring the machine in the appropriate state. Suppose for this example, that the voltage is ok when PowerSourceCtrl is activated. Then the **change event** isVoltageInRange() will only occur if

1. the battery voltage goes out of range and then
2. the voltage goes back into admissible range.

As a consequence, the state machine will never leave state BATTERY_VOLTAGE_NOT_OK, as long as the battery voltage is ok. This is certainly not the desired behaviour!

As we can see from this example, it is *mandatory* to provide these choice states and guard conditions for state machine initialisation in most cases.

Remember that all pseudo states must be left immediately. This requires that

- transitions leaving pseudo states must never be labelled by a trigger (absence of the trigger could block the transition), and
- the disjunction of all guard conditions associated with transitions leaving the choice state must always evaluate to `true` (this means that at least one transition must always be enabled).

5.4 Hierarchic State Machines

5.4.1 Hierarchy for Prioritised Behaviour

Let us analyse state machine PowerSourceCtrl (Fig. 5.1) further. The state BATTERY_VOLTAGE_OK is associated with a subordinate state machine shown in Fig. 5.2.

Hierarchic composite states for handling prioritised events

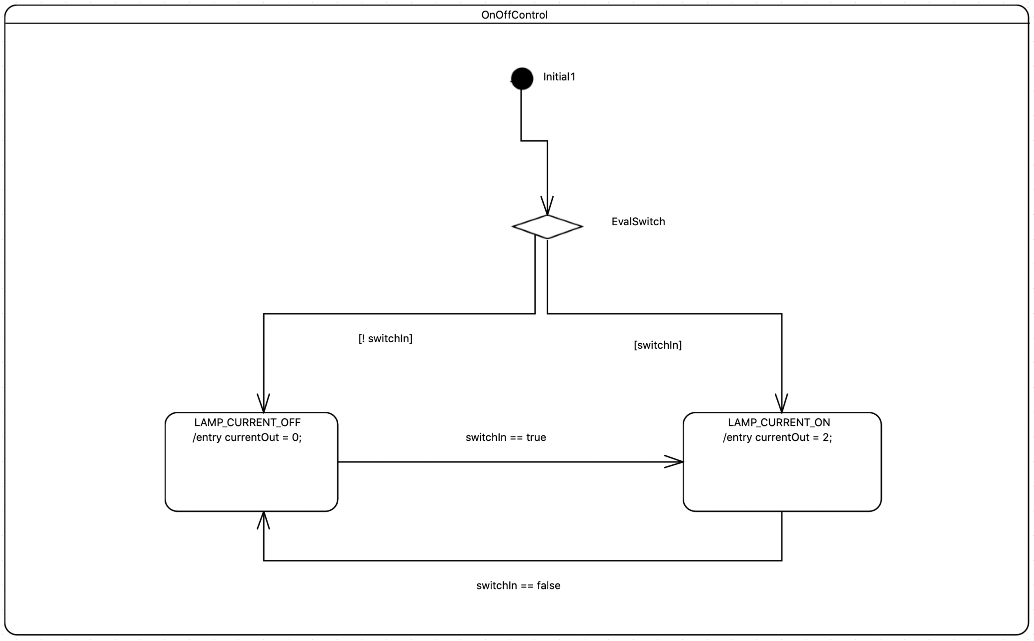


Figure 5.2: Subordinate state machine for state machine BATTERY_VOLTAGE_OK.

When entering composite state BATTERY_VOLTAGE_OK, the subordinate state machin `OnOffControl` is activated and branches immediately to one of the states LAMP_CURRENT_OFF or LAMP_CURRENT_ON, depending on the Boolean value on port switchIn. Then the normal behaviour

- Switch lamp current of when switch status changes to `false`,
- Switch lamp current on when switch status changes to `true`.

is executed by this state machine until the composite state BATTERY_VOLTAGE_OK is left via the higher-level transition from BATTERY_VOLTAGE_OK to BATTERY_VOLTAGE_NOT_OK.

It is correct syntax to use the same transition triggers with lower-level and higher-level transitions. However, this may be quite confusing, because

Don't use the same triggers on different levels

- The UML standard says that if this trigger occurs and both higher-level and lower-level transition are enabled, the *lower-level transition is taken*. This is usually confusing for people studying the model, since they associate higher priority with higher transition levels.

5.4.2 Composite States With Nested Regions

Recall from [1, 11.6][3] that a state machine state *s* becomes **composite** if we **nest** one or more state machines in one or more regions inside *s*. For very simple applications, this looks quite appealing, since the hierarchy of composite states and nested state machines becomes visible in one diagram. Please study the examples given in [1, 11.6.1, 11.6.2]. This, however, only works for very simple state machines, otherwise the diagram becomes extremely cluttered and unreadable. Therefore, we did not give any examples for composite states with nested regions in our turn indication model. Instead, we always use **submachines** as you can see in the turn indication model and read in the next paragraphs.

Use submachines instead of nested machines

5.4.3 Remarks on Submachine States

When associating a subordinate state machine with a state, the latter becomes **submachine state**. This term is slightly confusing: A submachine state is *not* a state of the subordinate state machine, but a **composite state** which *owns* or *is associated with* a submachine. Submachines may be associated with the

- do action,
- entry action, or
- exit action

[3] Yes, dear friends, you were supposed to read this.

of a state.

Guidelines for do/entry/exit

If the state machine is non-terminating, you should declare it as the do action of the state s: it will run as long as the state machine resides in s. If a transition leading away from s is triggered, the do action is aborted. State machines used as entry actions or exit action should terminate, because otherwise the state cannot be left again: an outgoing transition of s can only reach its destination state *after* the exit action has terminated. Recall that there exists a special symbol '×' for modelling termination states of a machine. Alternatively, the exit points explained below can be used.

Re-use of submachines

It is stated in [1, 11.6.5] (please study this section for further explanations) that submachines are intended for re-use. Please note that this kind of re-use is on a more localised level than re-use of blocks (that may have their owned classifier behaviour modelled by a state machine): state machines refer to ports or other properties of the owning block. Therefore, it is impossible to re-use them *in another block*. Therefore, submachines are typically applied if

- different hierarchic states of the same state machine should be associated with the same entry action, exit action, or do action, or
- simply because the submachine is too complex to be displayed as a nested region in a composite state.

Inter-level transitions

When displaying state hierarchy with nested state machines, it is allowed to model a transition from state of the nested machine to a state of the higher-level machine or vice versa. These transitions are called **inter-level transitions**, because the cross the boundaries between lower-level and higher-level state machines. Using inter-level transitions is rightfully frowned upon, since they are quite similar to `goto` statements on C++ programs and, therefore, make state machines extremely hard to verify.

Entry points, exit points, connection point references

In submachines, a more elegant substitute for inter-level transitions can be specified by means of entry points and exit points. An example of exit point usage is given in Fig. 5.3. Recall from our discussions during the tutorials, that submachine LRFLashing is associated as a do action with submachine state LRFlashing in the ControlLogic state machine. Submaschine LRFLashing handles the ON/OFF criteria for left/-right turn indication flashing, including tip flashing: When the submachine state LRFLashing is entered, the do action modelled by submachine LRFLashing starts operating, always beginning at the initial pseudo state. When entering successor state SendLR-

FlashCommand, the output port properties applyLeft and applyRight are set according to the position of the turn indication lever. Now we remember the tip flashing requirement: if the turn indication lever is put immediately back into neutral position, then the flashing on the left or right-hand side is only terminated after 3 ON/OFF cycles have passed. These three cycles have a duration 1980ms. During this time, we still remain in state SendLRFlashCommand, disregarding change events indicating that the turn indication lever is back in neutral position. After 1980ms, state SendLRFlashCommand is left, and the successor choice state is evaluated: if the lever position is still not in neutral, we transit to state TipFlashing Completed. Otherwise (guard condition `[LeverPositionINSW == NEUTRAL]` evaluates to `true`), we transit to the **exit point** ExitLRFlashing. This exit point can also be reached by a transition from TipFlashing Completed, when a change event occurs after the tip flashing time interval has passed, indicating that the lever is back in position neutral.

In order to specify what happens when a submachine transition reaches the exit point, a **connection point reference** is specified for submachine state LRFlashing. This is performed by selecting New Child $\rightarrow$ Connection Point Reference in the context menu of state LRFlashing. A good convention is to name the connection point reference just like the exit or entry point it is associated with. Then the Advanced menu of the connection point reference properties is opened, and the Exit property to be found there is linked to the exit point ExitLRFlashing of submachine LRFlashing. In the associated pop-up menu, you can only select exit points of an appropriate submachine. On the upper level, a transition is drawn from the connection point reference to a consecutive state. In the ControlLogic state machine, this is the choice state ifEmerOn.

Analogously, additional exit points and entry points (symbol ○) can be introduced for the sub machine and linked to connection points of the submachine state. Please read further details in [1, 11.6.5].

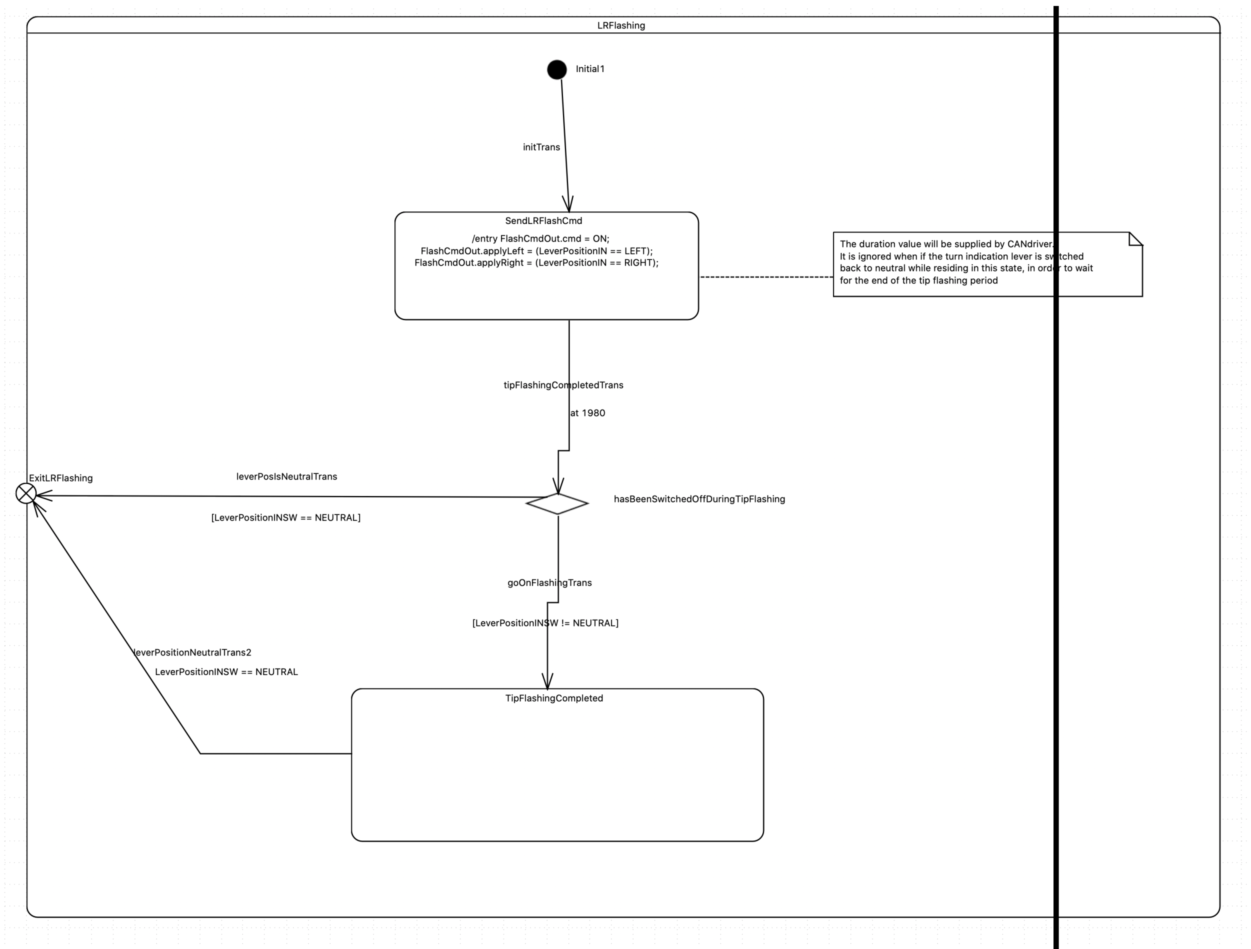

LRFlashing
Initial1
initTrans
SendLRFlashCmd
/entry FlashCmdOut.cmd = ON;
FlashCmdOut.applyLeft = (LeverPositionIN == LEFT);
FlashCmdOut.applyRight = (LeverPositionIN == RIGHT);
The duration value will be supplied by CANdriver.
It is ignored when if the turn indication lever is switched
back to neutral while residing in this state, in order to wait
for the end of the tip flashing period
tipFlashingCompletedTrans
at 1980
ExitLRFlashing
leverPosIsNeutralTrans
[LeverPositionINSW == NEUTRAL]
hasBeenSwitchedOffDuringTipFlashing
goOnFlashingTrans
[LeverPositionINSW != NEUTRAL]
leverPositionNeutralTrans2
LeverPositionINSW == NEUTRAL
TipFlashingCompleted

Figure 5.3: Submachine LRFlashing.

Chapter 6

Questions and Exercises

6.1 Questions

6.1.1 Hierarchic Composite State vs. Submachine State

A hierarchic composite state $\mathsf{s_c}$ has a lower-level state machine which is executed while the state machine resides in $\mathsf{s_c}$. Submachine states are also associated with a lower-level state machine. Why is it a good idea to have both constructs, instead of only allowing for hierarchic state machines?

For answering this question, it is advisable to study [3, 14.2.3.4.7].

6.1.2 Nested Blocks in Hardware Design?

Suppose you are using SysML for some HW design. Can you think of a situation where modelling the system with nested blocks woukld be appropriate?

6.2 Exercises

6.2.1 Block and State Machine Control Logic

The **control logic** of the turn indication function evaluates the interfaces TurnIndicationLever, EmergencySwitch, IgnitionSwitch, and BatteryVoltage. Based on these information and on its internal state, the control logic decides whether flashing should be activated on the left-hand side, right-hand side, or on both sides. This decision is communicated to a component driving the CAN bus by means of an output port containing Boolean Properties leftOn, rightOn. Note that the control logic does not handle the on-off phases, nor does it take care of tip flashing. It just determines the required flashing status by setting the port properties in the appropriate way.

1. Create a block ControlLogic which becomes part of the rear controller.
2. Associate a state machine as classifier behaviour for this block, so that the requirements concerning decisons about flashing on left, right, or both sides are correctly modelled.

   It is recommended to use both hierarchic and parallel states to model this behaviour, because the machine will become quite complex otherwise.

6.2.2 CAN Output Control and Flash Cycles

As specified in the requirements, a second block **CANCtrl** needs to be allocated in the rear controller which sends the appropriate CAN bus messages to all slave recipients (ON commands with duration and applyLeft, applyRight information, OFF commands when required). This component is responsible for the proper flashing cycles and for tip flashing. On an input port, the Boolean Properties leftOn, rightOn are received from the control logic. As output port, the CAN interface is used.

Create a state machine as classifier behaviour of this block in accordance with the applicable requirements.

6.2.3 Reuse of the PowerSource and LampControlSlave Blocks

In one of our lectures, it has been explained how the blocks PowerSource and LampControlSlave can be re-used to control all turn indication lamps in the four controllers. This has been exemplified in the current turn indication model using the two door controllers. Integrate the blocks PowerSource and LampControlSlave into the front and rear controller, together with appropriate configuration blocks PinProgramBlockLeft and PinProgramBlockRight.

For the rear controller, you may assume that the CAN bus has a **loop back** capability: the messages sent away over the CAN software port of block

CANCtrl may be fed back into the CAN input port CANinSW of the two LampControlSlave instances allocated in the rear controller.

Bibliography

[1] Sanford Friedenthal, Alan Moore, and Rick Steiner. *A Practical Guide to SysML, Third Edition: The Systems Modeling Language*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2014.

[2] David Harel and Amnon Naamad. The statemate semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, October 1996.

[3] Object Management Group. OMG Unified Modeling Language (OMG UML), version 2.5.1. Technical report, OMG, 2017.