

# INSTITUT FÜR INFORMATIK

## **SCCharts: The Mindstorms Report**

Steven Smyth, Christian Motika,  
Alexander Schulz-Rosengarten, Sören Domrös,  
Lena Grimm, Andreas Stange, and  
Reinhard von Hanxleden

Bericht Nr. 1904

December 2019

ISSN 2192-6247



CHRISTIAN-ALBRECHTS-UNIVERSITÄT  
ZU KIEL

Institut für Informatik der  
Christian-Albrechts-Universität zu Kiel  
Olshausenstr. 40  
D – 24098 Kiel

## **SCCharts: The Mindstorms Report**

Steven Smyth, Christian Motika,  
Alexander Schulz-Rosengarten, Sören Domrös, Lena Grimm,  
Andreas Stange, and Reinhard von Hanxleden

Bericht Nr. 1904  
December 2019  
ISSN 2192-6247

E-mail: {ssm, cmot, als, sdo, lgr, aas, rvh}@informatik.uni-kiel.de

Technical Report

# Contents

<b>1. Introduction</b>	<b>1</b>
1.1. Lego Mindstorms . . . . .	1
1.2. SCCharts . . . . .	3
1.2.1. Transformation Complexity . . . . .	4
1.2.2. The KIELER SCCharts Editor . . . . .	8
1.3. Contributions & Outline . . . . .	9
<b>2. The Mindstorms Tasks</b>	<b>10</b>
2.1. Seat Belt and Dome Light Controller . . . . .	10
2.2. A Simple Measuring Tape . . . . .	11
2.3. System Under Development . . . . .	12
2.4. Barcode Reader . . . . .	14
2.5. Pathfinder . . . . .	16
<b>3. Survey Results</b>	<b>18</b>
3.1. Survey Setup . . . . .	18
3.2. Language Aspects . . . . .	20
3.2.1. Deterministic Behavior . . . . .	21
3.2.2. Programming Paradigms . . . . .	22
3.2.3. Problem Solving . . . . .	23
3.2.4. Language Difficulty . . . . .	25
3.2.5. Modularity . . . . .	26
3.2.6. Project Revisions . . . . .	26
3.3. Feature Aspects . . . . .	28
3.3.1. Basic Transition Features . . . . .	28
3.3.2. History, Suspension, and Actions . . . . .	29
3.3.3. Concurrency, Declarations, and Data Types . . . . .	30
3.3.4. Additional Extended Features . . . . .	31
3.3.5. Future Features . . . . .	32
3.4. Tooling Aspects . . . . .	34
3.4.1. Model Creation and Debugging . . . . .	34
3.4.2. Further Tooling Aspects . . . . .	35
<b>4. Related Work</b>	<b>37</b>
4.1. Railway Projects . . . . .	37
4.1.1. Railway Demonstrator . . . . .	37
4.1.2. Train Controlling . . . . .	38

4.1.3. Practical Labs . . . . .	38
4.1.4. Survey . . . . .	39
4.2. Teaching Synchronous Languages . . . . .	39
4.2.1. Survey . . . . .	39
4.3. Raceyard . . . . .	40
<b>5. Wrap-Up</b>	<b>42</b>
5.1. Conclusion . . . . .	42
5.2. Future Work . . . . .	42
5.2.1. Object Orientation . . . . .	42
5.2.2. Model Checking . . . . .	43
<b>A. SCCharts Survey</b>	<b>48</b>
<b>B. SCCharts Short Survey</b>	<b>63</b>
<b>C. SCCharts Tutorial</b>	<b>65</b>
<b>D. SCCharts Cheat Sheet</b>	<b>103</b>

# List of Figures

1.1.	An assembled NXT . . . . .	2
1.2.	The Core SCCharts transformation matrix . . . . .	3
1.3.	The SCCharts base model for complexity approximations . . . . .	4
1.4.	Complexity ratio for expressions and valued objects . . . . .	6
1.5.	Timeline of the KIELER SCCharts Editor development . . . . .	7
1.6.	Simulation perspective of the KIELER SCCharts editor . . . . .	8
2.1.	A seat belt and dome light controller . . . . .	11
2.2.	Measuring tape examples . . . . .	12
2.3.	A controller for subtask of task 3 . . . . .	13
2.4.	Barcode encoding . . . . .	14
2.5.	Barcode reader simulation visualization . . . . .	14
2.6.	An example for an barcode reader . . . . .	15
2.7.	The pathfinder mat . . . . .	16
3.1.	Language preferences . . . . .	21
3.2.	General determinism . . . . .	21
3.3.	Deterministic concurrency . . . . .	21
3.4.	Sequentiality . . . . .	22
3.5.	Separate timing & functionality . . . . .	22
3.6.	Solving abstract problems . . . . .	23
3.7.	Solving low-level problems . . . . .	24
3.8.	Understandability . . . . .	25
3.9.	Simplicity . . . . .	25
3.10.	Composability . . . . .	26
3.11.	Maintainability . . . . .	26
3.12.	Debugging . . . . .	27
3.13.	Essential transition features . . . . .	28
3.14.	History and local action features . . . . .	29
3.15.	Concurrency, declarations, and data types . . . . .	30
3.16.	Additional extended features . . . . .	31
3.17.	Future features . . . . .	33
3.18.	Tool quality . . . . .	34
3.19.	Model creation & debugging . . . . .	35
3.20.	Tooling aspects . . . . .	35
4.1.	Railway installation and first SCCharts railway controller . . . . .	37
4.2.	Using SCCharts Models in Simulink to Model an Electronic Control Unit . . . . .	40

5.1. Inheritance and method regions in SCCharts . . . . .	43
5.2. Model Checking for SCCharts . . . . .	44

## **Abstract**

SCCharts are a visual language proposed in 2012 for specifying safety-critical reactive systems. This is the second SCCharts report towards the usability of the SCCharts visual language and its KIELER SCCharts implementation. KIELER is an open-source project which researches the pragmatics of model-based languages and related fields.

Nine case-studies that were conducted between 2015 and 2019 evaluate the pros and cons in the context of small-scale Lego Mindstorms models and similar projects. Participants of the studies included undergraduate and graduate students from our local and also external facilities, as well as academics from the synchronous community. In the surveys, both the SCCharts language and the SCCharts tools are compared to other modeling and classical programming languages and tools.

# 1. Introduction

The visual programming language SCCharts was presented in 2012. It was specifically designed for *safety-critical systems* that interact with their environment continuously [vHDM<sup>+</sup>14]. Safety-critical systems usually have long development cycles. Before deployment, it is common to pass several iterations of analysis, design, construction, and simulation phases. First prototypes are also commonly deployed to *demonstrators* which show the capabilities of a particular product before they are implemented in a live environment where the product often is used for decades.

The first SCCharts case-study [SMSR<sup>+</sup>15] illustrates how the participants of that study used SCCharts to control a railway installation. At the end of the project, the controller was able to control all existing eleven trains concurrently with dynamic schedules. The participants evaluated SCCharts as a programming language and the Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER) SCCharts tool chain. The feedback was used to increase the performance, stability and usability. Also, some new features were added to the language and tooling repertoire to streamline the development process. The SCCharts implementation is nowadays known as *KIELER SCCharts Editor*.

The participants of the case-studies presented in this report mostly used Lego Mindstorms as demonstrator. Analogously to the railway installation, it is a great benefit when teaching real-time principles of cyber-physical systems if students are able to experiment with real hardware environments/physics. Especially mechanical inaccuracies and real-time requirements are more challenging on hardware demonstrators than in software simulations. These differences become apparent to the students if experienced first hand.

For the sake of completeness and as comparison, the results presented here also cover related surveys that were conducted in the context of another railway project and advanced synchronous language lectures.

## 1.1. Lego Mindstorms

LEGO<sup>®</sup> Mindstorms<sup>®</sup><sup>1</sup> was designed to combine creativity and problem solving into one easy-to-program unit. The LEGO bricks allow use-case-specific configuration and design to enable the robot to fulfill a particular task. Participants of the Embedded Real-Time lectures from our local department used the Mindstorms NXT for all tasks, which are specified and explained in Chap. 2.

---

<sup>1</sup><https://lego.com>





Figure 1.1.: An example of an assembled NXT. This NXT is not fit to fullfil most of the tasks of the lecture. Available at: [www.lego.com](http://www.lego.com)

The NXT is available since 2006 and is based on an ARM processor<sup>2</sup> and has three actor ports and four sensor ports as well as an USB port. Moreover, it can connect via Bluetooth. The LEGO Mindstorms Education NXT base set includes three motors, which have rotation sensors, two touch sensors, a sound sensor, an ultrasonic sensor, and a light sensor, which also includes a lamp. Three lamps from the previous Mindstorms generation and enough LEGO bricks to design different robots as well as instructions on how a robot can be assembled are also included. An assembled NXT robot can be seen in Fig. 1.1 as illustration. However, the depicted robot is not able to fulfill most of the tasks of the lecture.

The leJOS<sup>3</sup> framework is a replacement firmware for the NXT that allows to program the Mindstorm in Java. LeJOS builds upon the TinyVM<sup>4</sup>, a small virtual machine for Java primarily used in embedded systems. The memory footprint of the OS is only about 10Kb with objects only having an overhead of 4 bytes each. However, the low memory usage goal of the TinyVM results in a field limitation of 255 each, meaning that there

---

<sup>2</sup><https://www.arm.com>

<sup>3</sup><http://www.lejos.org>

<sup>4</sup><http://tinyvm.sourceforge.net>



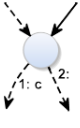
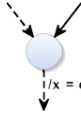
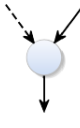
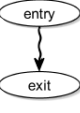
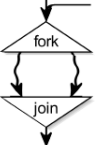
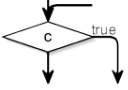

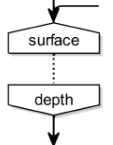
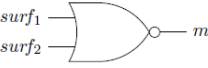
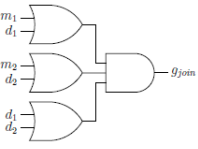
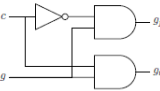
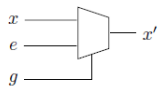

	Region (Thread)	Superstate (Concurrency)	Trigger (Conditional)	Effect (Assignment)	State (Delay)
SCCharts					
SCG					
Data-Flow Code	$d = g_{exit}$ $m = \neg \bigvee_{surf \in t} g_{surf}$	$g_{join} = (d_1 \vee m_1) \wedge (d_2 \vee m_2) \wedge (d_1 \vee d_2)$	$g = \bigvee g_{in}$ $g_{true} = g \wedge c$ $g_{false} = g \wedge \neg c$	$g = \bigvee g_{in}$ $x' = g ? e : x$	$g_{depth} = \text{pre}(g_{surf})$
Circuits					

Figure 1.2.: The Core SCCharts transformation matrix

are at most 255 classes, with each one including up to 255 variables. The maximum array length per dimension is also 255. Variables are not aligned in memory for space efficiency reasons. These limitations are great for teaching programming of real-time and embedded systems, because such systems often only have scarce resources available.

Since the KIELER compiler is able to generate Java code, SCCharts can be used to program an NXT via leJOS. However, the dataflow-based code generation algorithm of the KIELER SCCharts tools generates a classical netlist, which tends to use more variables for program guards than a manually-written Java program. The variable usage is a concern in the actual compiler and has been optimized during the more recent lecture runs.

## 1.2. SCCharts

The SCCharts language is divided into two parts: *Core SCCharts* and *Extended SCCharts*. Core SCCharts comprise all elementary language patterns, which are necessary to compile, whereas Extended SCCharts include more complex elements, often regarded as syntactic sugar, to create more compact programs. All Extended SCCharts are compiled to Core SCCharts before being processed further by the low-level part of the compiler. The five core patterns are depicted in Fig. 1.2. The main patterns are (from

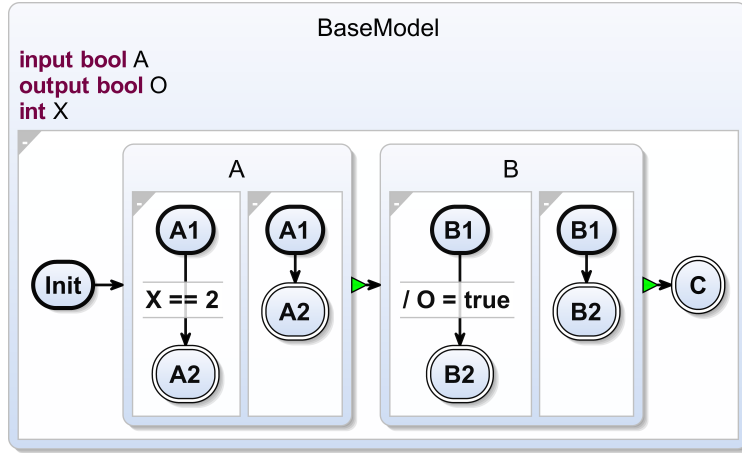


Figure 1.3.: The SCCharts base model for complexity approximations

left to right in the figure) *regions*, which resemble threads, *superstates*, which model concurrency, *trigger*, which guard transitions, *effects*, which emit effects if a transition becomes active, and *states*, which consume time. These patterns are translated one-to-one into an control-flow graph. The predominant compilation approach in KIELER then creates a netlist. The specific guard formulae are shown in the **Data-Flow Code** row. Furthermore, it is possible to translate these netlists directly to circuits, which is shown in the row below.

### 1.2.1. Transformation Complexity

Motika showed that every Extended SCCharts model feature can be transformed into a semantically equivalent Core SCCharts model [Mot17]. However, the complexity of the resulting Core SCCharts model varies with each extended feature, which has a great effect on the structure of the control-flow graph and on the variable count of the netlist. Hence, some features have a greater impact on the viability w.r.t. the NXT limitations mentioned in Sec. 1.1. In order to measure the complexity of specific extended features, we created a *base model*, which can be seen in Fig. 1.3. This base model only uses features included within the scope of Core SCCharts. It is constructed such that extended features can be added without introducing new structural model elements. Thus, only the complexity resulting from the use of one designated extended feature is added to the base model. The complexity values of the base model and the model including the extended feature allow the calculation of the complexity ratio  $C_{\text{ratio}}$  as follows

$$C_{\text{ratio}} = \frac{f(P_{\text{feature}})}{f(P_{\text{base}})}$$

We measure two different domains: the amount of variables  $f_{\text{vars}}$  and the complexity of expressions  $f_{\text{exp}}$ . Both values are measured in the final sequentialized control-flow graph resulting from the netlist-based compilation approach. The complexity value  $f_{\text{vars}}$

Model	Valued Objects		Expressions	
	$f_{vars}$	$C_{vars}$	$f_{expr}$	$C_{expr}$
<b>base</b>	<b>9</b>	<b>1.00</b>	<b>58</b>	<b>1.00</b>
const	8	0.89	58	1.00
connector	9	1.00	58	1.00
actionEntry	9	1.00	60	1.03
actionExit	9	1.00	60	1.03
initialization	9	1.00	60	1.03
static	9	1.00	60	1.03
abortStrong	9	1.00	94	1.62
abortWeakImmediate	9	1.00	97	1.67
abortStrongImmediate	9	1.00	110	1.90
abortWeak	10	1.11	90	1.55
actionDuring	11	1.22	89	1.53
complexFinalStates	11	1.22	94	1.62
abortConditionalTerminationImmediate	11	1.22	115	1.98
history	11	1.22	126	2.17
deferred	12	1.33	92	1.59
signalPure	12	1.33	99	1.71
abortConditionalTermination	12	1.33	101	1.74
pre	13	1.44	106	1.83
suspension	13	1.44	154	2.66
signalValued	14	1.56	167	2.88
complexFinalStatesHierarchical	15	1.67	264	4.55
countdelay	16	1.78	165	2.84

Table 1.1.: Complexity values and ratios using the two different measurement approaches

counts the amount of used variables including the variables for the guards of the netlist. The value  $f_{exp}$  evaluates the expressions used by counting the amount of operands and operators. More precisely, the following formula is used for the calculation:

$$f_{exp}(P) = \sum_{e \in E} \begin{cases} 1 & type(e) = Literal \\ 1 & type(e) = ObjectReference \\ 1 + \sum_{e' \in exprs(e)} f_{exp}(e') & type(e) = OperatorExpression \end{cases}$$

whereas  $E$  is defined as the set of all root expressions included in the program  $P$ . The function  $exprs(x)$  is defined as the shallow set of expressions of the expression  $x$ . As an example, for an expression  $e$  with  $e = (a + b) - c$  the function is defined to return  $exprs(e) = \{a + b, c\}$ . Hence, the function  $exprs$  does not traverse the expression tree recursively.

We created a new model for each extended feature that we wanted a measurement of. Consequently, the two different complexity measurement approaches are performed on those models. The results are summarized in Tab. 1.1. The first line holds the values for the base model which serves as the denominator for the calculation of the specific ratio. The subsequent lines are sorted by the result ratio for the valued objects approach. This

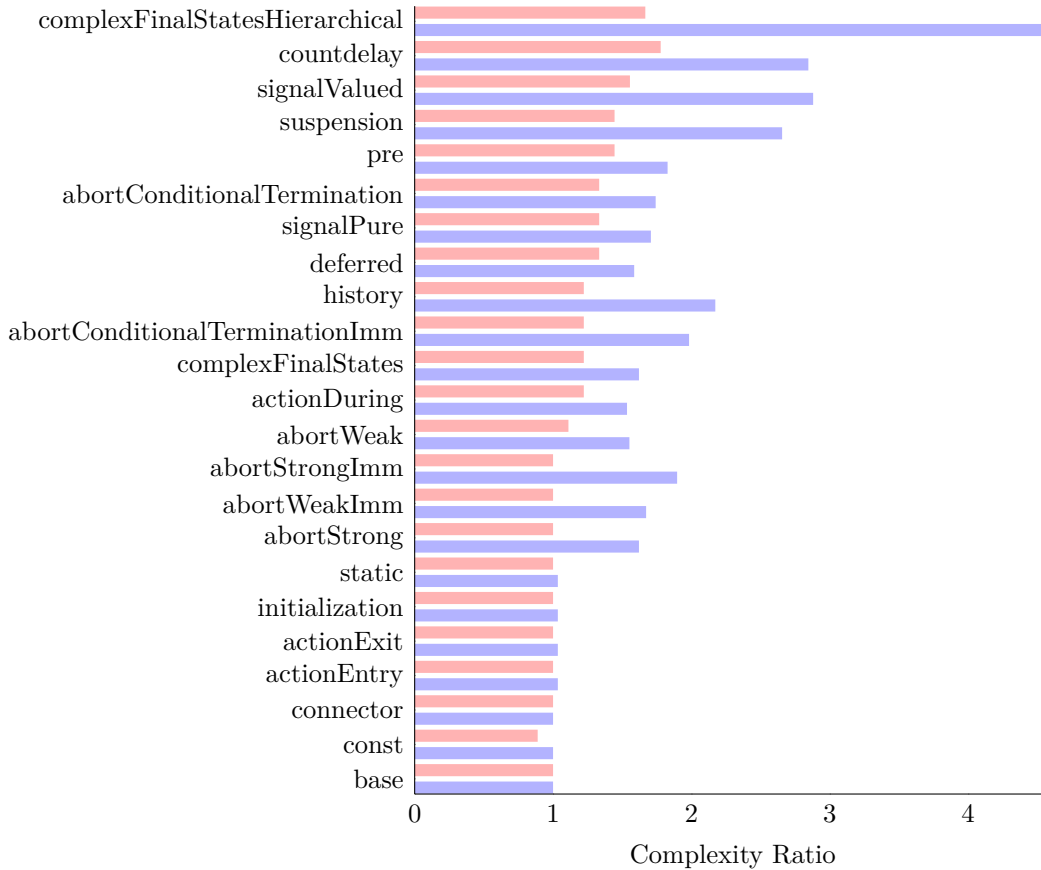


Figure 1.4.: Complexity ratio for expressions (blue) and valued objects (red)

establishes a relation to the Lego Mindstorm field limitations.

The results presented in the table show that there are certain transformations that introduce a great amount of complexity regardless the measurement approach. In general, the ratio for expression complexity lies within  $[1, 4.55]$  and the ratio for the amount of valued objects is within  $[0.89, 1.78]$ . The ratio below 1 is achieved using the constants declaration which allows to optimize code generation and thus saves one variable. However, both approaches lead to a similar conclusion. The transformation of hierarchical complex final states, count delays and valued signals seems to introduce the most complexity regardless of the measurement approach. In conclusion, for modeling Lego Mindstorms, some of the extended features might expand the original model in such ways that they are not viable with respect to the limitations discussed in Section 1.1.

Note that the values presented here significantly depend on the nature of the base model. More or less hierarchy may lead to different peaks in the ratio. Especially combinations of different extended features may result in an unexpected growth of the model. The complexity results illustrated here give an idea of the implications of different transformations, but they need to be generalized in order to make a more elaborated statement on the effects of different transformations. However, with growing and more complex models the measurements are expected to further increase. Since the complexity

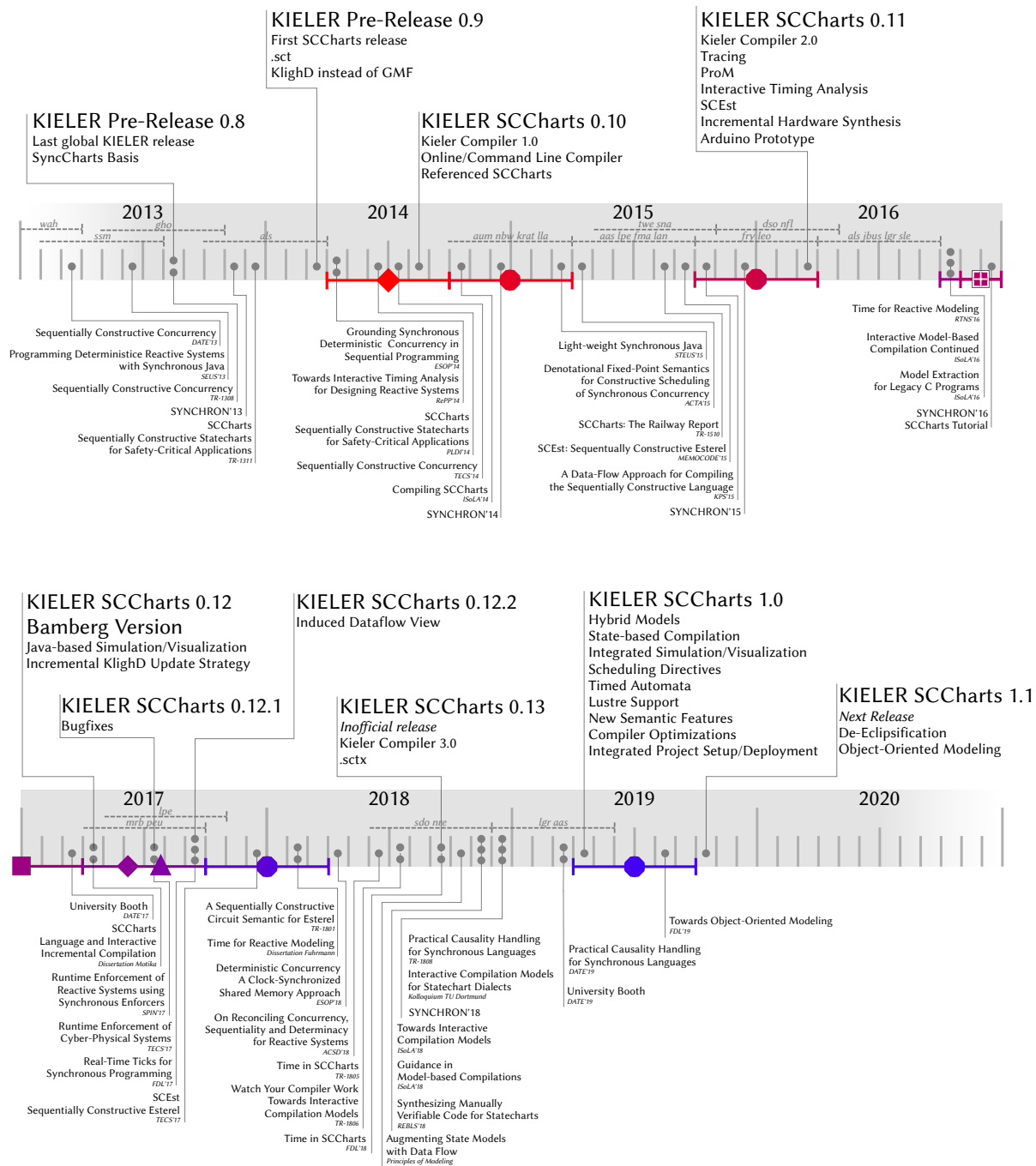


Figure 1.5.: Timeline of the KIELER SCCharts Editor development

is hidden from the modeler by design to enable them to create more compact models, this kind of transformation complexity is coined *internal complexity*.

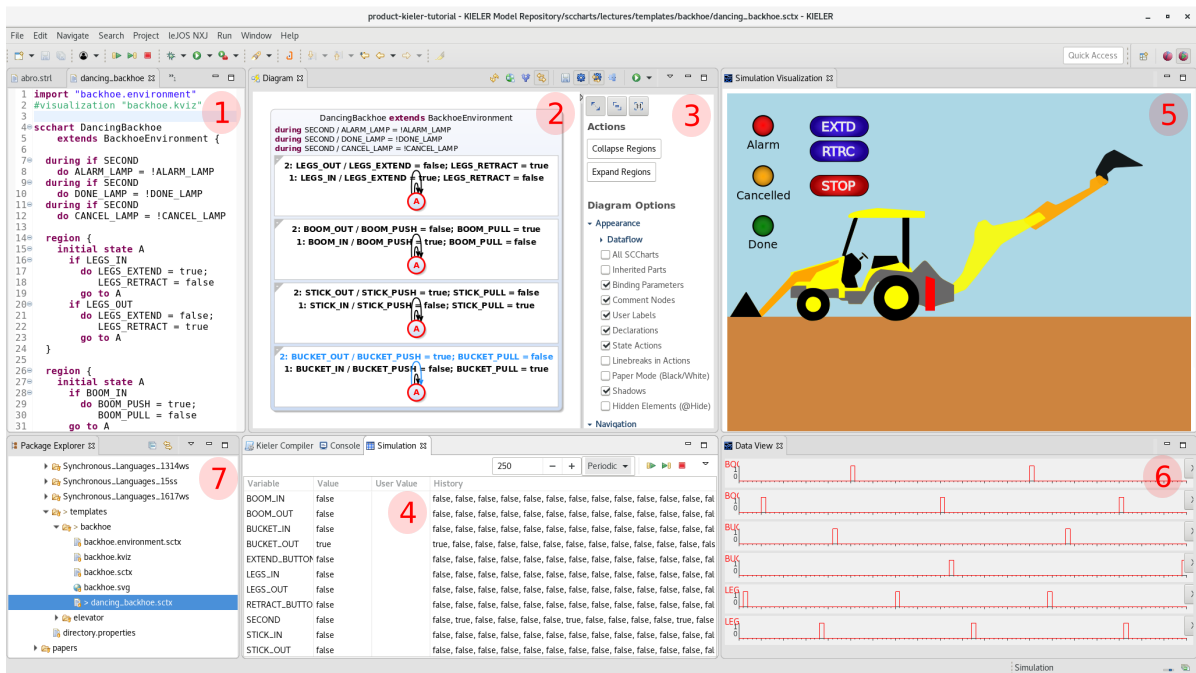


Figure 1.6.: Simulation perspective of the KIELER SCCharts editor

## 1.2.2. The KIELER SCCharts Editor

The KIELER SCCharts Editor contains the most complete implementation of the SC-Charts language. The overall development timeline of the editor is depicted in Fig. 1.5. It shows the releases, including milestones, above and the related publications below the timeline. The surveys, which will be discussed in Chap. 3, are drawn along the bottom border. The survey's symbol indicates the kind of the associated project. A diamond  $\blacklozenge$  marks a railway project, a circle  $\bullet$  a Mindstorms project, a square  $\blacksquare$  a synchronous lecture, a rectangle  $\blacktriangle$  a external project, and a cross  $\boxplus$  the survey conducted during the Synchronous Workshop. Related theses are depicted inside the timeline. Overall, there were seven dedicated releases (excluding maintenance releases) of the SCCharts editor and 44 publications and presentations related to the project. The eighth release is scheduled for late 2019.

The conducted user studies led to a series of improvements, such as code optimizations [Bus16,SSRvH18d] and more sophisticated debugging capabilities [WSRSvH18,SS-RvH18a,Gri16]. In parallel, we developed a fully modular, model-based compiler framework [SSRvH18c], which can be used to compile programs, perform simulations, and deploy compiled artifacts. A customizable integrated template engine manages wrapper code generation, which is often necessary to embed the generated model code in the used environment. Without changing the code generated for the models, templates can be easily adjusted to support cognate environments, such as, e.g., standard C and Arduino C. Together with new pragmatic solutions for handling large models, such as the

Eclipse Layout Kernel's<sup>5</sup> (ELK) *Label Management* [Sch19], the tooling improved over time.

Fig. 1.6 shows a running simulation of a backhoe task, in which the backhoe digs a hole into the ground. The controller program can be edited in the text editor ①. The automatically synthesized diagram is depicted in ②. States that are active at the end of a simulation step are depicted in red. Taken transitions are shown in blue. Several synthesis options can be toggled in ③ to bring different parts of the program in focus instantly. The environment variables that are used for communication with the controller are shown in the simulation view ④. The pace of the simulation is also controlled here. A graphical representation of the whole environment is shown in ⑤. The backhoe has several statuses, indicated by three lamps in the upper left, and three buttons that control the machinery. Its stick movement can be seen in the animation. Timelines with the variables' histories are shown in a dedicated data view ⑥. Finally, the modeler can switch to different projects or load additional data in the project explorer ⑦.

### 1.3. Contributions & Outline

Chap. 1 already showed the complexity measurements in Sec. 1.2.1 and the whole development cycle of the SCCharts Editor's development, including key improvements, in Sec. 1.2.2.

In Chap. 2 we describe the usual tasks that the participants of our internal case studies had to solve to give an overview over the complexity of the programs and also to show what kind of introductions the participants got before answering the survey's questions. Afterwards,

- we present the results of five years of SCCharts surveys, which were filled out after working a short amount of time with the KIELER SCCharts tools. The whole survey can be found in Appendix A. All results are shown in Chap. 3 ordered by category. In the surveys, the participants should rate different aspects of the SCCharts languages in comparison to other popular or related languages. Additionally, they should grade the quality of the KIELER SCCharts tools.
- The tooling category in Section 3 also includes the data of a shortened SCCharts survey (see Appendix B) handed out to academics at the Synchronous Workshop in Bamberg 2016.
- Simultaneously, we discuss the results and give recommendations to further development.

Section 4 shows related projects. We conclude in Section 5.

---

<sup>5</sup><https://www.eclipse.org/elk>



## 2. The Mindstorms Tasks

Four of the nine surveys, which will be discussed in Chap. 3, were conducted after the Real-Time and Embedded Systems class held at the Department of Computer Science at Kiel University. During the lecture several tasks were used to give a step by step introduction to SCCharts and how to program the LEGO Mindstorms with them. As an example, the tasks from the last lecture in the summer term '19, which mostly appeared in the preceding iterations, are presented here to give an overview over the preparation the survey participants had before they filled out the questionnaire.

The goal throughout the semester is to develop a more extensive program to fulfill some real-time task by writing a controller that enables the Mindstorm to follow a black line as described in Sec. 2.5. This is realized as a contest between the participants in teams of two in which they try to develop a program that can solve the task correctly and in a short time. In the previous lecture runs the number of used ticks was optimized by taking the concept of dynamic ticks into account (cf. timeline in Fig. 1.5). As preparation for the final contest several tasks are introduced to get familiar with the sensors and actuators of the robot, timing, and the interaction with the environment and how such an interaction can be simulated.

The first task, which is not mentioned here, is to install and set up the leJOS framework. Installing leJOS allows KIELER to use the corresponding commands to compile and deploy to the Mindstorms robot. Since the installation of leJOS is different for all OSs and cannot be completely tested for all OS variants and versions, this task allows to solve problems before the real tasks begin.

### 2.1. Seat Belt and Dome Light Controller

The goal of the seat belt and dome light controller task is to teach the students how the different sensors, actuators, and their macros work. Moreover, this task introduces real-time constraints to SCCharts and gives an example of how two different models can be combined in SCCharts. The task requires to model a seat belt and a dome light controller for a car given the following specification:

The dome light is turned on as soon as any door is opened. Consider two different doors. It stays on at least 3 but not 4 seconds after all doors are shut. Assume that the automobile provides the input *open* when the first door is opened and *closed* when the last open door is closed, specifically the input  $x(t) \in \{open, closed, absent\}$ . Once the engine is started, a beeper sound and a red light warning indicate if there are passengers that have not buckled their seat belt. The beeper stops sounding after 30 seconds, or as soon the seat belts are buckled, whichever is sooner. The warning light is on all the

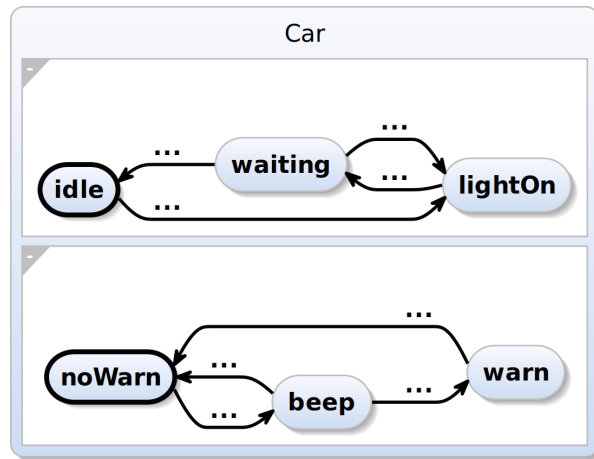


Figure 2.1.: A seat belt and dome light controller (see Sec. 2.1)

time the seat belt is unbuckled.

The task includes hints what sensors and actuators can be used to model a controller from this specification. Most participants modeled their controller via two different regions with a state for the status of the seat belt and dome light controller. This results in a model with two regions with three states each as seen in Fig. 2.1.

## 2.2. A Simple Measuring Tape

The Mindstorm cannot directly infer the distance it traveled. However, the rotation of a motor since the start of the program is available. This task teaches students fundamentals about motor rotation sensors and their relation to physical distances. It is a preparation for driving the robot in the path finder task.

The participants should develop a measuring tape that displays the distance a motor rotated in some real measuring unit. This task gives the participants freedom to make their measuring tape as sophisticated as they like.

Student solutions range from a one state solution that uses the motor rotation and diameter of the used tire to calculate the distance traveled in, e.g., millimeter to a complex model with several regions and states that allows to switch between different units, can be reset, and pretty prints the current value instead of logging the current distance on the screen as seen in Fig. 2.2. This task can be easily solved via a dataflow model, but students tend to model it via control flow using one single state. A reason for that might be that dataflow regions were only recently introduced to the students.

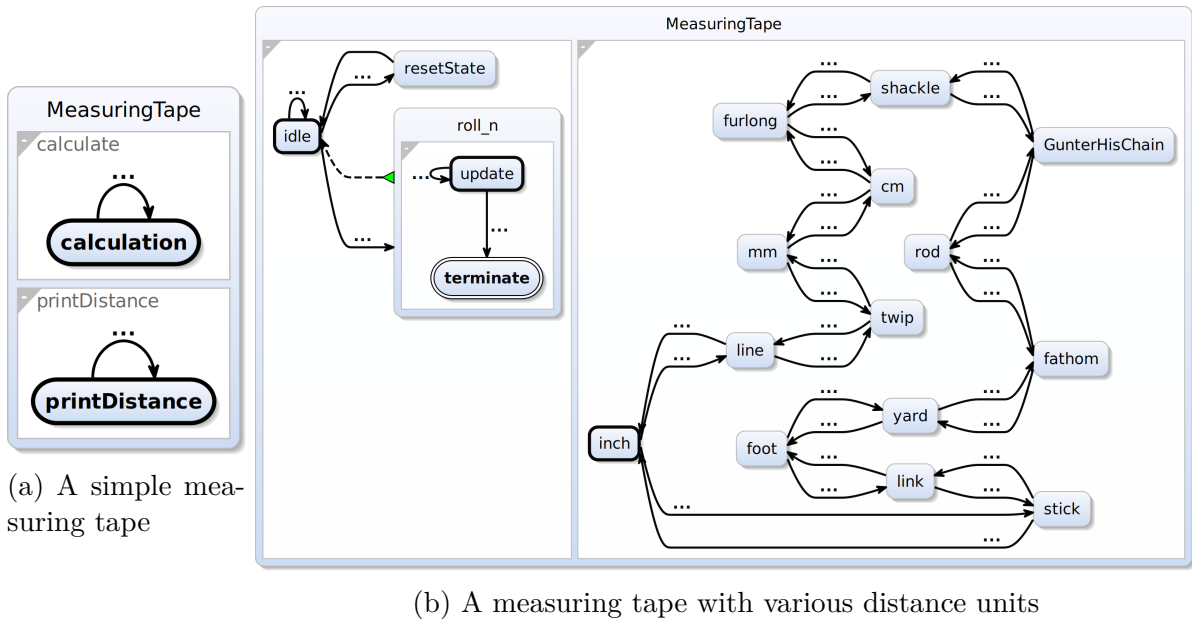


Figure 2.2.: Measuring tape examples

### 2.3. System Under Development

The next step is to use the acquired knowledge about timing and the distance-rotation relation to develop a controller for a car (or robot) with a corresponding environment to test the behavior. This task is divided in three subtasks:

- Develop an SCChart for a car controller which set the speed and rotation of the vehicle to drive a route in the shape of an eight. Moreover, an environment for this car should be modeled, which receives speed and rotation as an input and outputs the x and y coordinate as well as the direction the car faces in degree.
- Add the seat belt and dome light controller from Sec. 2.1 to the car controller in a modular way. This can be done by adding the regions used in the previous task to this controller or by referencing the already modeled SCChart from Sec. 2.1. Moreover, the environment should be extended to generate inputs for the seat belt and dome light controller.
- The last part requires the participants to add an abstraction layer for their controller. This abstraction layer calculates a speed for the left and right motor of the Mindstorm robot from the existing speed and rotation provided by the already developed controller. The students are free to make more changes to their controller if necessary. The final goal is to have a Mindstorm robot that can drive a route in the shape of an eight.

The developed controller can be tested in KIELER via the simulation feature. The controller and the environment of the subtasks can be combined into one SCChart to

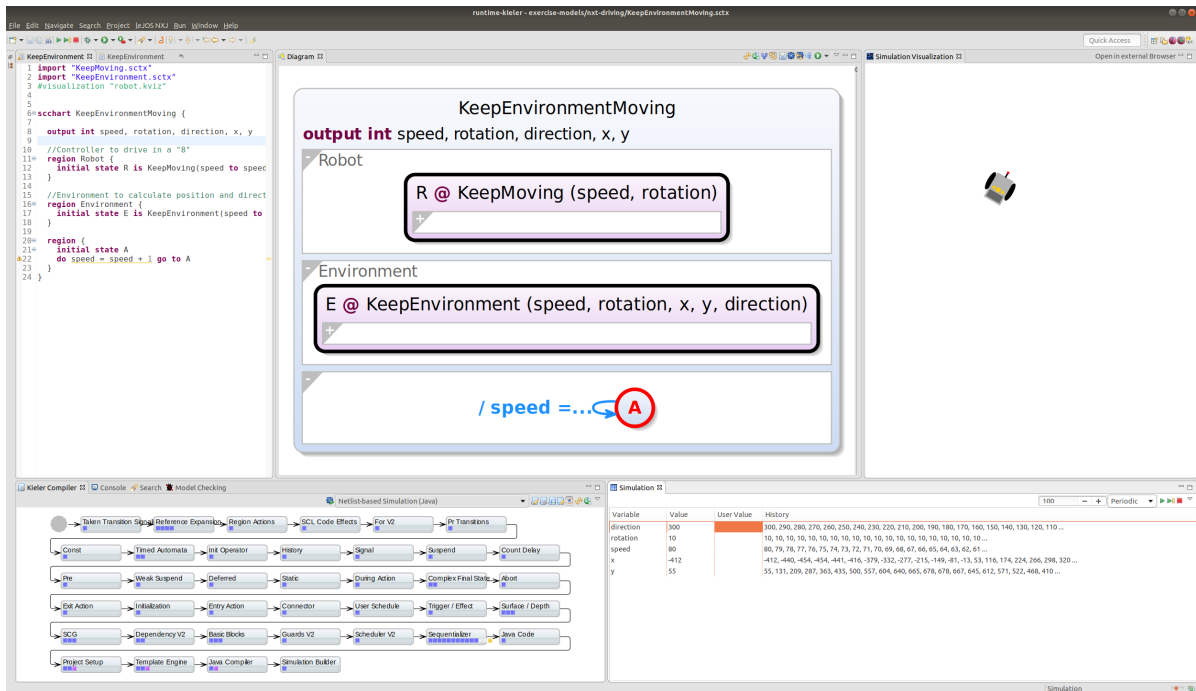


Figure 2.3.: A controller for the first subtask of task 3 (see Sec. 2.3) is simulated with the corresponding environment inside KIELER. On the top left the editor is shown. Next to it is the automatically synthesized diagram with taken transition highlighting. To the right of the diagram the simulation visualization view displays an image of the robot using the current position and orientation calculated by the environment. In the bottom right is the simulation view that displays the corresponding inputs and outputs in form of a table.

simulate and test the controller. This allows to visualize the car or robot and its position and direction via an SVG that is manipulated via JavaScript as seen in Fig. 2.3. Since not all JavaScript code might be supported by the Eclipse internal mini browser, an external browser can be used instead. For all following tasks this kind of visualization is provided to the participants of the lecture.

The solutions to this problem follow different approaches. Some student groups used the current speed of the robot to calculate the time it should take to drive a circle. Some used the rotation of each motor to calculate the position and orientation of the robot. Unlike the Mindstorm EV3 (the newest Mindstorm generation), the NXT does not include a sensor to determine its orientation. Therefore, students have to use timing or the rotation sensors in order to calculate the orientation.

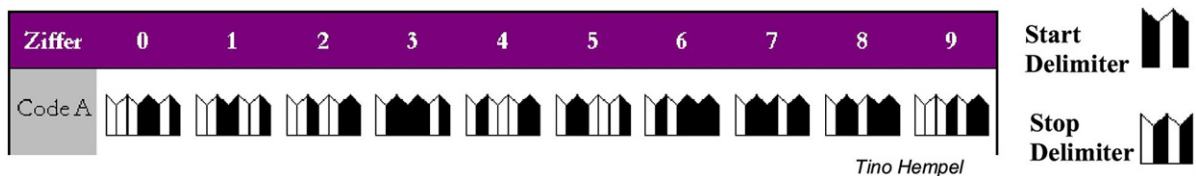


Figure 2.4.: Each barcode begins with a start delimiter and ends with an stop delimiter. Each number is encoded by seven black or white bars which correspond to the European Article Number (EAN).



Figure 2.5.: An example how the SVG for simulating the barcode reader looks like. The corresponding JavaScript also allows the visualization to work as an environment by setting input variables for the controller in form of the light value.

## 2.4. Barcode Reader

This task is deemed the most difficult and students generally have two instead of one week time to work on it. It pushes the variable limit of the NXT and requires to optimize the developed SCChart towards this property. The main goal is to combine real-time aspects and light-level detection via the light sensor to prepare for the final pathfinder contest.

The challenge of this task is to develop a controller that enables a Mindstorm to drive over one or several barcodes and read the corresponding numbers according to the encoding described in Fig. 2.4. This task also provides a simulation visualization via an SVG as seen in Fig. 2.5, which enables the students to test all corner cases of their program.

The real barcodes consist of 5mm wide bars and include four numbers. The first three correspond to the encoded number, the last is a check digit that ensures correctness of the read digits. A barcode is valid if  $d_4 = 9 - ((d_1 + d_2 + d_3) \bmod 10)$  for a barcode consisting of  $d_1$ ,  $d_2$ ,  $d_3$ , and  $d_4$ . Part of the task is the mentioned validation check for the read digits.

Depending on the realization, it might not be possible to develop a state machine that determines the number because of the variable limitation as discussed in Sec. 1.1. A state for each possibly read bar combination usually results in too many variables. The variable usage can be optimized by using an array or a bit vector instead of a state

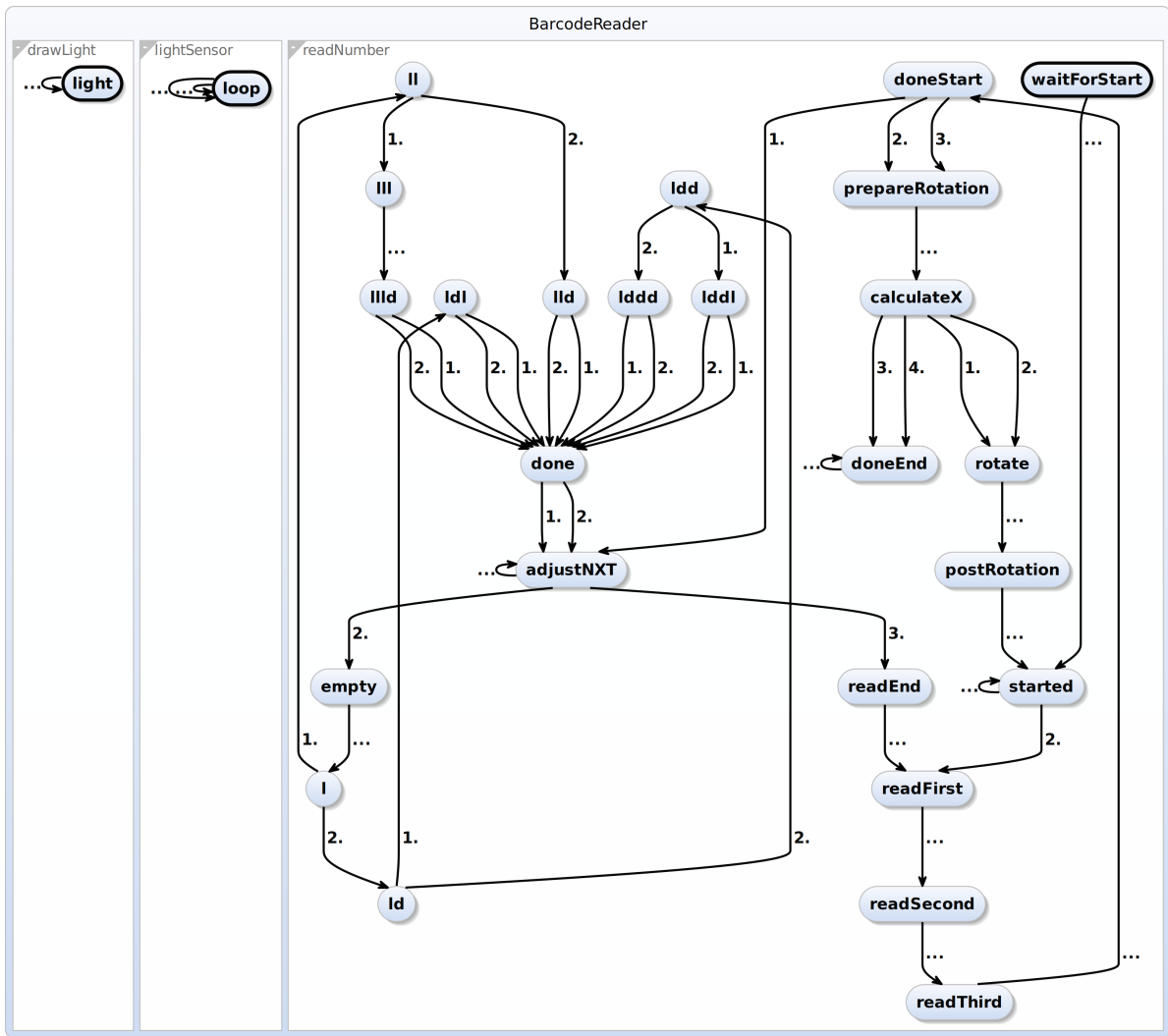


Figure 2.6.: The barcode reader is divided into three regions. The *drawLight* region draws to the display of the Mindstorm, the *lightSensor* region reads the light value and categorizes it in light or dark, and the *readNumber* region evaluates the light values. This controller has more functions than the task requires, since it also rotates according to the read number on the barcode and continue to search for a new barcode in that direction. This model of a barcode reader was developed in winter term 15/16. Models developed in the summer term 19 tend to be bigger, since the code generation was optimized with respect to used variables.

machine. Alternatively, not all bars of a digit have to be checked, since their encoding is redundant. An exemplary controller that does not check all bars of a digit can be seen in Fig. 2.6. Student models often include a setup phase to calibrate the light sensor and the start delimiter is often used for to measuring the length of each line. The motor rotation can be used to obtain the same effect. The specified length of 5mm was used by students to recognize when bars start and stop. However, this solution is not robust.

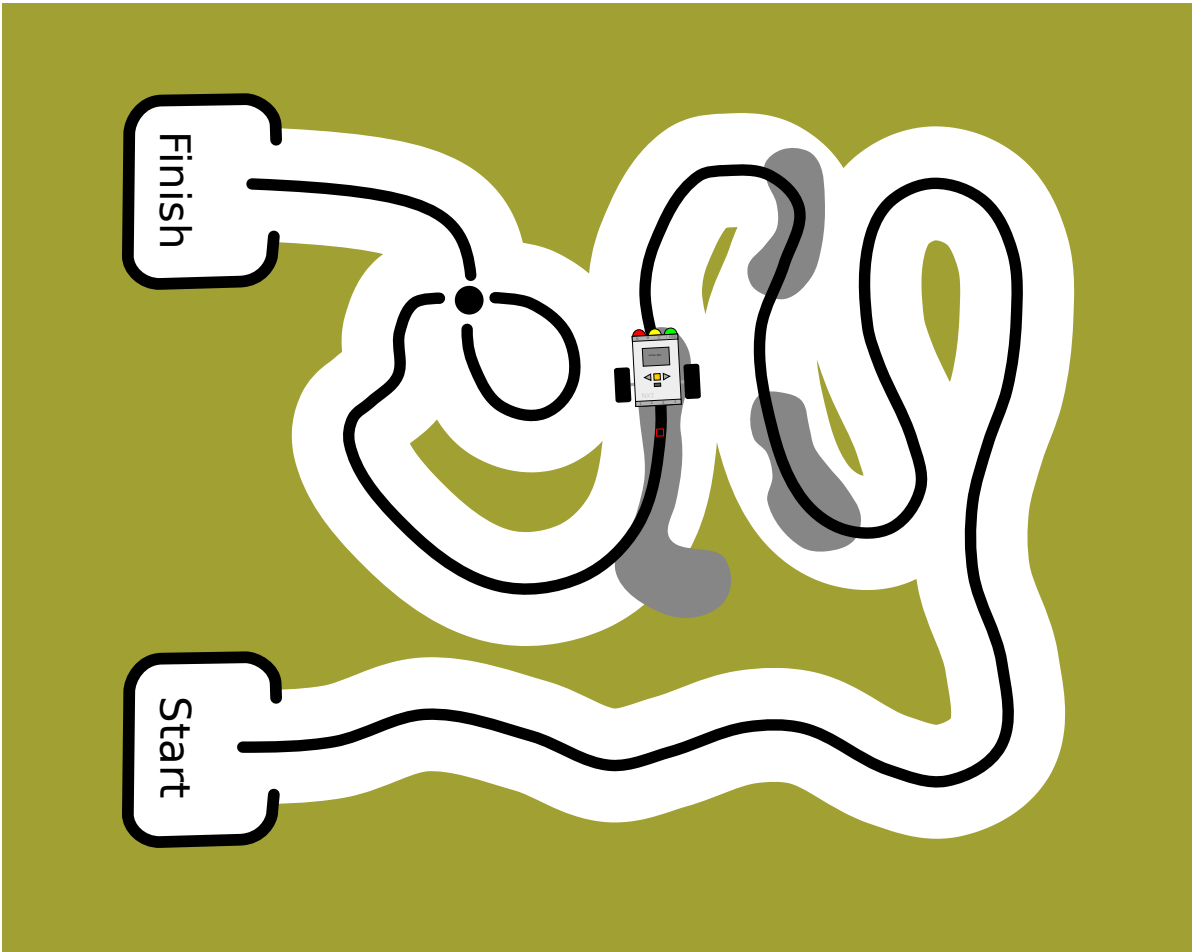


Figure 2.7.: The pathfinder mat as used in the simulation visualization.

Inaccuracies and different drive paths make this solution prone to errors.

## 2.5. Pathfinder

The results of the pathfinder task are presented in form of a contest in which the participants compete with each other regarding time and, optionally, used ticks. The goal is to follow a black line. The line has a thickness of approximately 1.5cm from start to finish and the mat has the dimensions of approximately 1.4m width and 1.8m height. As before, a corresponding visualization is provided as can be seen in Fig. 2.7, which can be used to simulate the controller in KIELER. The physical mat has the following main obstacles and challenges which might not be present in the simulation:

- The gray areas might confuse the light sensor and the robot might go off track. This cannot be emulated in the simulation, since the light value is very precise while simulating and the one on the physical mat might be noisy.

- Between the gray areas and the loop the real mat has white writing on the line. This makes it more difficult to recognize whether the robot is actually on the line or not. As before, this cannot be correctly emulated in the simulation.
- The loop has white areas on entry and exit. This can result in skipping or being trapped in the loop if the wrong black line is recognized. It is recommended to test the robot's path at the crossing on the physical mat, because the scaling in the simulation may suggest a different behavior. The loop's crossing must be passed straight.
- Moreover, the robot must be able to turn fast enough, to successfully pass curves. This is again a timing and speed constraint that cannot be emulated.

The deviations between the SVG and the real mat also teach the students that simulations do not always fit the real world exactly. Nonetheless, they help to develop a working prototype.

There are several different approaches to solve this task:

- Most groups developed a controller with a state for driving left and one for driving right. Additional states for finding the line again after losing it and calibrating the light sensor are also present in most models. These solutions usually have between two and eight states.
- The most advanced controllers model a PID controller and drive on the edge of the line. Groups that model their pathfinder via a PID controller have often one of the fastest NXTs.

Calibration of the light sensor or a good threshold value for the brightness are essential to many groups and need to be fine tuned.



## 3. Survey Results

The survey (see Appendix A) was filled out for the first time by the participants of the railway project in the summer term 2014 [SMSR<sup>+</sup>15]. It was handed out several subsequent times: All students of the the real-time and embedded systems lectures in the terms winter 14/15, winter 15/16, winter 17/18, and summer 19 as well as students from the synchronous lecture in the winter term 16/17 and the railway project in the summer term 2017. Furthermore, students from the University of Auckland completed the survey at the end of their embedded systems class in 2017 and the participants of the Synchron Workshop in 2016 were asked to fill out the survey during the workshop in Bamberg in December 2016. This chapter evaluates the results of these surveys and compares them to the previous results presented elsewhere [SMSR<sup>+</sup>15]. As before, there are three parts that were considered:

- 1. Language Aspects:** In this part the surveys ask general questions about SCCharts and comparisons to other languages. The results are presented and discussed in Section 3.2.
- 2. Feature Aspects:** This part asks the participants about SCCharts features and their relevance towards their project. The results are presented and discussed in Section 3.3.
- 3. Tooling Aspects:** In the third part, the participants were asked to give feedback about the KIELER SCCharts implementation. The results are presented and discussed in Section 3.4.

### 3.1. Survey Setup

The evaluation in this chapter summarizes the results of the different project groups that all filled out the same survey. All participants were asked to fill out their survey at the end of their particular project. In the following, the groups are distinguished by the marker shape and color in the diagrams. As described in Sec. 1.2.2, the shape of the marker determines the type of project: A diamond  $\blacklozenge$  marks a railway project, a circle  $\bullet$  a Mindstorms project, a square  $\blacksquare$  a synchronous lecture, a rectangle  $\blacktriangle$  a external project, and a cross  $\blacksquare$  the survey conducted during the Synchronous Workshop. They are always displayed in chronological order from left to right, additionally indicated by the red-to-blue color gradient.

**First Survey — Railway Project, summer 2014** ( $\blacklozenge$ ) The first survey of this form was handed out at the end of the railway project in the summer term 14. There were 7

participants that filled out the survey and all were pursuing a master's degree in Computer Science. The results are already discussed in the preceding SCChart's report series [SMSR<sup>+</sup>15].

**Second Survey — NXT, winter 2014/15 (●)** The second survey was handed out at the end of the real-time and embedded systems lecture in the winter term 14/15, where the participants solved various tasks with the NXT Lego Mindstorm as described in Chap. 2. There were 21 participants. All of them were Computer Science bachelor students.

**Third Survey — NXT2, winter 2015/16 (●)** The third survey was handed out at the end of the real-time and embedded systems lecture one year later. Compared to the preceding year, the tasks in this year were more challenging w.r.t. the SCCharts models. In particular, the participants reached the limit of the model sizes that could be uploaded onto the Mindstorm, as described in Sec. 1.1. There were 34 students. Most of the participants pursued a bachelor's degree in Computer Science with a few exceptions already finished their bachelor's studies.

**Fourth Survey — Synchron 2016, winter 2016/17 (■)** The fourth survey was handed out to the participants of the Synchron Workshop 2016 in Bamberg<sup>1</sup>. As an introduction to SCCharts, a interactive tutorial (see Appendix C and the accompanying SCCharts Cheat Sheet (see Appendix D) was conducted in 1.5h. As last part of the tutorial, the participants were ask to solve the pathfinder task similar to the one described in Chap. 2. After the tutorial, a shortened survey (Appendix B) could be given back optionally. Overall, we received 8 responses from the Synchron Workshop audience.

**Fifth Survey — Synchronous Lecture, winter 2016/17 (■)** The fifth survey was conducted at the end of the Synchronous Languages lecture in the winter term 2016/17. While the tasks during the lectures differed from the ones that had to be completed during the Embedded Systems classes, we included the results of this survey for the sake of completeness. 17 participants answered the questionnaires. They were mostly students from the Master's degree program.

**Sixth Survey - Railway Project 2017, summer 2017 (◆)** For the same completeness reason, we included the results of the second railway project from the summer term 2017 in this report. While there were only 5 survey participants, they were all graduate students similar to the fifth survey. Together with the 17 students from the fifth survey and the 7 students from the first railway project, 29 graduate students participated overall.

**Seventh Survey — External, summer 2017 (▲)** The seventh survey was handed out to a group of students from the University of Auckland. 8 undergraduate students that attended Prof. Roop's Embedded Systems class<sup>2</sup> participated in summer 2017. As

---

<sup>1</sup><https://www.uni-bamberg.de/gdi/synchron-2016>

<sup>2</sup><https://unidirectory.auckland.ac.nz/profile/p-roop>

a member of the synchronous community, Prof. Roop is familiar with synchronous languages and their principles. The course did not receive any active support from the KIELER team during the term. We are very proud that our tools contributed to teaching at an external department and thankful for the feedback we received.

**Eighth Survey — NXT3, winter 2017/18** (●) The eighth survey was handed out at the end of the real-time and embedded systems lecture in the winter term 2016/17 with similar tasks as in the lectures before. Most of the 12 students, who participated in the survey, pursued a bachelor’s degree in Computer Science.

**Ninth Survey — NXT4, summer 2019** (●) The last survey covered in this report was handed out at the end of the real-time and embedded systems lecture of the summer term 2019. The tasks were similar to the previous embedded systems courses. 23 participants, again mostly bachelor students in Computer Science, handed in their survey responses.

Overall 135 participants took part in this SCCharts survey over the course of 5 years. While we experienced a growth in the stability of the developed SCCharts tools, the different survey groups did not have the exact same starting points and hence, there is no concrete reference frame. However, we think that the studies are comparable, especially if relative ratings within a question between different groups match. Also, if the relative ratings of groups with a low number of participants get backed-up by the results of larger survey groups. So, when talking about relative trends, we did only compare the distinct survey groups compared to themselves as the different case studies did not have a common reference frame.

In general in all questions that compare different languages, the synchronous and model languages taught at the department were chosen, namely SCADE, Esterel, SyncCharts, SCCharts, and Ptolemy. Additionally, two mainstream imperative languages, C and Java, and one functional language, Haskell, were selected. However, because the first three language choices were relatively unknown to the audience because the second and third survey was conducted with bachelor students, the following results will only include SCCharts, Ptolemy, C, Java, and Haskell. As the students of the Auckland group were not familiar with Ptolemy and Haskell, they are excluded from these comparisons. For Ptolemy, the students of the Real-Time and Embedded Systems classes used the latest stable version of Ptolemy II at the particular times.

## 3.2. Language Aspects

**Description** As a general question, the participants were asked which languages they find suited for the given tasks (the tasks from the embedded systems class are described in Chap. 2). The results depicted in Fig. 3.1 show that SCCharts as well as C and Java were found suitable. Ptolemy still scored OK whereas Haskell was situated in the lower segment by the students for the tasks focused on designing hardware controller.

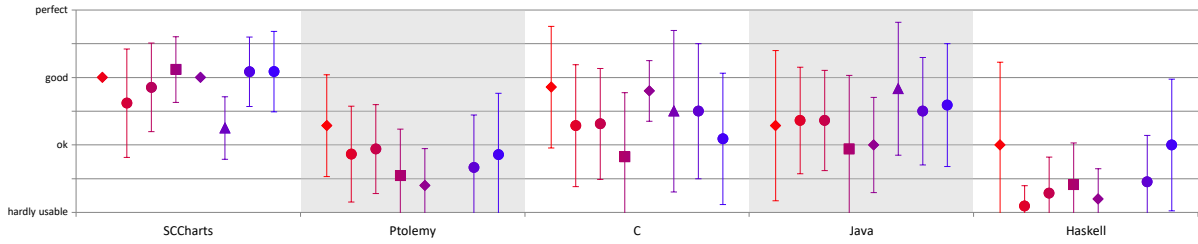


Figure 3.1.: Language preferences

**Results** These results confirm the voting from the first survey [SMSR<sup>+</sup>15] and positions SCCharts to one level with C and Java when it comes to cyber-physical systems. Hence, SCCharts are deemed capable to model software for hardware controllers.

### 3.2.1. Deterministic Behavior

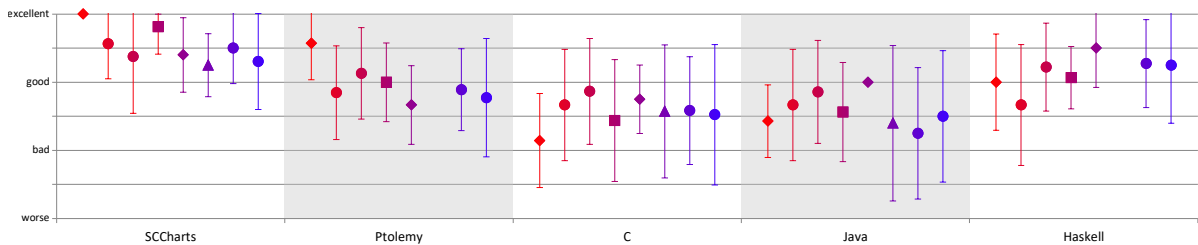


Figure 3.2.: General determinism

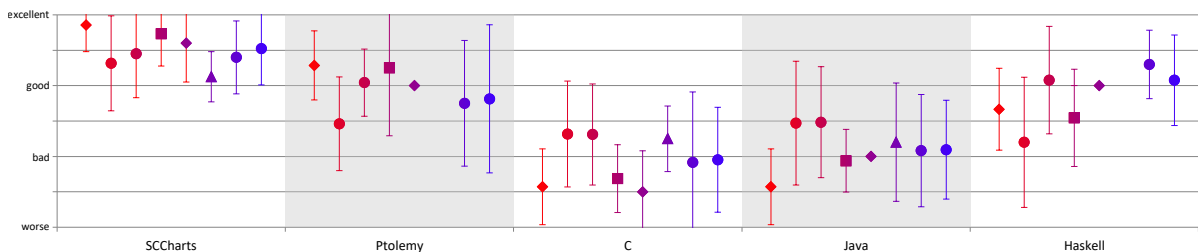


Figure 3.3.: Deterministic concurrency

**Description** The next two figures depict the results of the questions about how much effort is needed to achieve deterministic behavior. Fig. 3.2 shows the answers regarding the overall capability of a language to be deterministic: Is it hard for a modeler or programmer to reach determinism and is the deterministic behavior understandable? Fig. 3.3 presents results for deterministic concurrency, arguably the hardest part in classical programming languages when it comes to determinism. The main question here is how much effort is necessary to avoid *race conditions*.

**Results** In both cases, the participants confirm the trend which was observed in the 2014 survey [SMSR<sup>+</sup>15]. Naturally, archiving deterministic behavior is easier with synchronous languages, including SCCharts, because they are deterministic by design. Especially when it comes to race conditions, synchronous languages excel. The rules that synchronous Models of Computation (MoCs) enforce make it relatively simple to write deterministic and concurrent programs. However, as a consequence the set of programs that are accepted as *constructive* is more restricted than in classical languages. Also, the participants rated concurrent programming in purely functional languages such as Haskell higher, which might be rooted in the fact that these languages employ a side-effect free paradigm.

Note that in contrast to the results of the 2014 survey [SMSR<sup>+</sup>15], the voted grades of the classical programming languages are not as bad as before. Also, Fig. 3.2 looks pretty similar to Fig. 3.3 hinting at deterministic concurrency being a key enabler to archive deterministic programs.

The modeler’s experience also plays a role. The ratings of the graduate students are more extreme than the ratings from the undergraduate students. Usually, it is necessary to teach race condition problems to undergraduate students, whereas concurrency in synchronous languages can be taught relatively easy.

### 3.2.2. Programming Paradigms

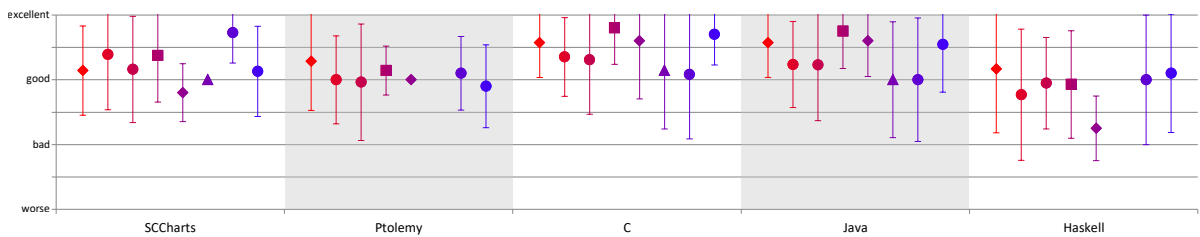


Figure 3.4.: Sequentiality

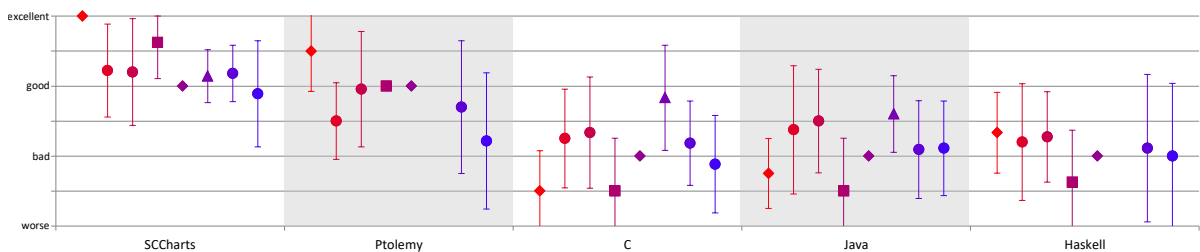


Figure 3.5.: Separate timing & functionality

**Description** The next two questions cover sequentiality in programs, with the results displayed in Fig. 3.4, and the separation of timing and functionality, with the results depicted in Fig. 3.5. Naturally, writing sequential programs is not that hard in classical imperative programming languages and is a common drawback of synchronous languages. Here, due to the classical synchronous MoC expressing sequential control flow becomes hard for a modeler. Closing this gap is one key challenge of SCCharts.

Especially safety-critical systems often require a separation of functionality and concrete timing. Worst-case execution time (WCET) analyses can prove that each reaction of a program can be asserted. This is another strong field for synchronous languages because they are designed this way.

**Results** As before, the results confirm the results of the first survey. For sequentiality the results are nearly the same. The two synchronous languages only have slight disadvantages compared to the imperative languages. However, despite the fact that SCCharts use the sequential constructive MoC, in comparison to Ptolemy, the ratings are only slightly better with an upwards trend towards the end of the study. Maybe expressing sequentiality is more of a convenience feature in synchronous modeling and not that much of a general issue. Nonetheless, we still believe that the ability to express sequential pattern helps traditional programmers to transition to the synchronous paradigm, which will be discussed further in Sec. 3.2.4.

The results in Fig. 3.5 are less extreme than in 2014, but the trend is similar: Synchronous languages are ahead, mostly because they are designed this way. However, in classical programming languages it seems to be notoriously hard to separate the functionality of the actual timing of the program. Particularly, the students from the larger railway projects confirm this. Also, Ptolemy was rated a mark worse than in 2014, which might have its cause in fewer Ptolemy exercises as the focus of the lecture shifted more towards KIELER as the IDE matured. Nonetheless, with many different *directors* and MoCs in Ptolemy, it might be a bit more confusing to separate timing and functionality, even if Ptolemy is more powerful than SCCharts w.r.t. MoCs.

### 3.2.3. Problem Solving

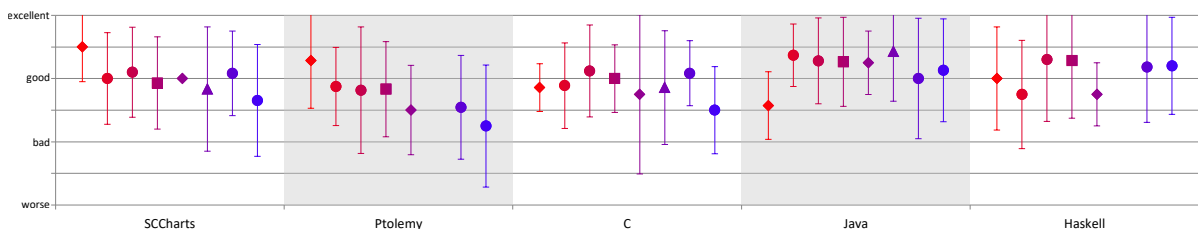


Figure 3.6.: Solving abstract problems

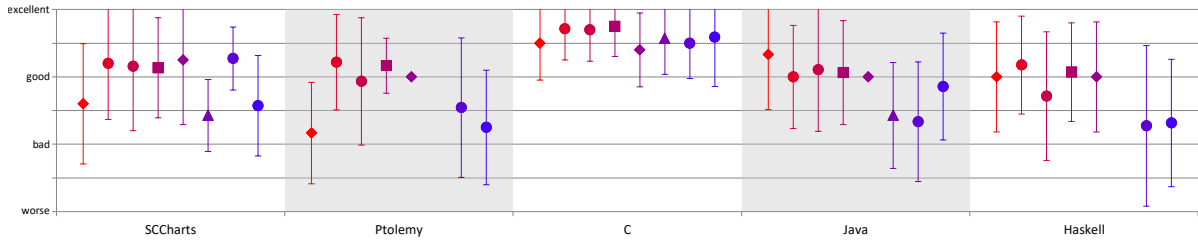


Figure 3.7.: Solving low-level problems

**Description** This section presents the results of the questions which language performs better in solving *abstract*, Fig. 3.6, and/or *low-level*, Fig. 3.7, tasks. Abstract problem solving focuses on the ability to find solutions to problems, such as keeping the pathfinder on track (see Chap. 2). Contrary, low-level problem solving gives solutions to a particular low-level problem, such as how can I power up the next three track segments of the railway installation (see Chap. 4).

**Results** Towards solving abstract problems, the difference is marginal when compared to C and worse for the synchronous languages when compared to Java. The exception here, is the first railway project. In contrast, the results for the low-level ratings are reversed. Here, synchronous languages rate considerably better for solving low-level task for the Mindstorms tasks and worse during the railway project. However, C is the undisputed language in this comparison when it comes to low-level tasks.

One reason for these ratings may be the task sizes within the different projects. In 2014, the participants were tasked to create one big system with a concrete C interface. During the Mindstorms classes, several smaller models were developed and deployed distinctively to the Mindstorms. Therefore, picking SCCharts to model the complex railway system in an abstract way might be a good choice while dealing with the C interface in this context can become cumbersome. However, for smaller models, such as the ones for the Mindstorms, these strengths may become disadvantages, because of the MoCs restrictions. The hardware interaction is not that complicated even on model level.

Furthermore, new additions to the KIELER SCCharts tools, such as those described in Sec. 1.2.2, have improved the capability to interact with hardware directly. The need to use *hostcode* calls was reduced dramatically. Hence, the modeler is free to concentrate on the actual problem even though when dealing with low-level issues.

It should be investigated further if the project size directly influences the language preference and efficiency of the developer and how modern model-based developing environments can improve the developer's experience if this is the case.

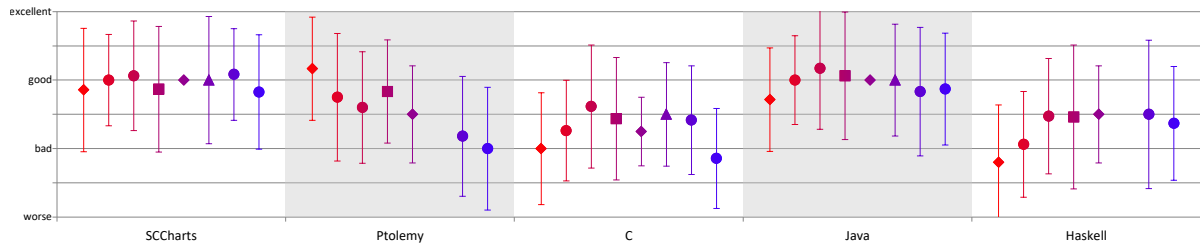


Figure 3.8.: Understandability

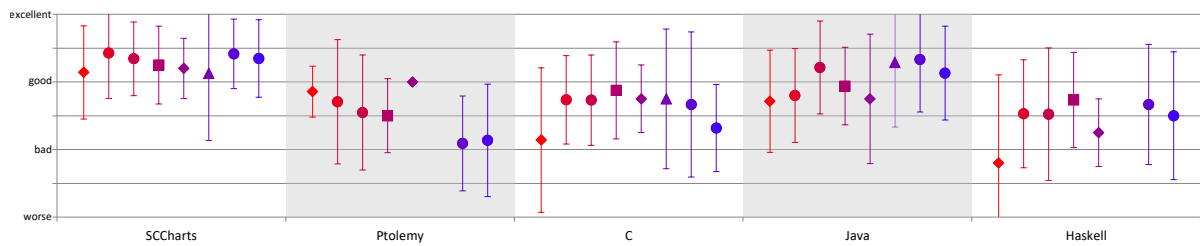


Figure 3.9.: Simplicity

### 3.2.4. Language Difficulty

**Description** Fig. 3.8 illustrates the ratings on how difficult it is to read and understand models or programs. Especially when working with large models and/or in teams, it is crucial to keep models understandable.

Fig. 3.9 shows the votes on how simple it is to learn a certain language in order to fully utilize its features to create comprehensible models or programs. A language that is simple to learn and whose more advanced features are easy to comprehend helps to improve the efficiency of the developers and to maintain an understandable state of the project.

**Results** In general, the results confirm the previous results. Even though the project sizes are different, the understandability and simplicity of the languages are nearly equal. In both categories SCCharts was rated to be as good or better than Java, which has the highest rating compared to the other mainstream languages. In comparison to Ptolemy, SCCharts also scored better, but again, here, we see the reasons mostly in the unfamiliarity of the participants and the higher complexity of the Ptolemy language. The project size does not seem to have a large impact on understandability or simplicity of the SCCharts models. Also, despite the fact that the external student group did not receive direct support if problems occurred with KIELER, their results confirm the results of the department's local students, and hence, supports the fact that SCCharts is indeed simple to learn and to understand.



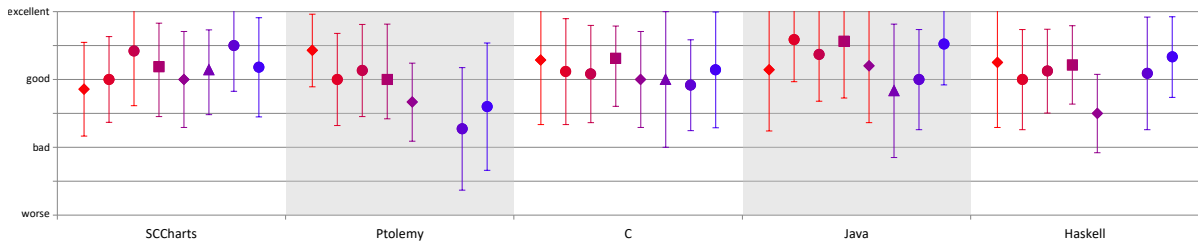


Figure 3.10.: Composability

### 3.2.5. Modularity

**Description** Fig. 3.10 depicts the answers to the question how easy it is to compose multiple models/programs to form larger projects. The ability to divide a project into smaller parts enables enhanced structuring and facilitates team development. It also is a key enabler for defining interfaces within the project and for the reuse of components.

**Results** With the exception of the last surveys w.r.t. Ptolemy, all languages were rated good w.r.t. composability. In modeling languages structuring and navigating projects is not a trivial task. The features of the editor, such as navigating hierarchies through new windows like in Ptolemy or SCADE, shape the developers experience. It is important for these languages to be on par with classical languages, which basically use the structure mechanisms of the file system or an extension thereof.

In the first railway study, SCCharts were rated worse in comparison to the other languages. However, the relative SCCharts results improved in the later studies, where SCCharts was deemed more on par with the other languages. Here, project size, but also KIELER improvements (see Sec. 1.2.2) such as the implementation of an own import mechanism and the abolishment of the Eclipse natures, might be important factors for the ratings' improvements in later studies; especially, compared to the separate window approach used in languages, such as Ptolemy.

### 3.2.6. Project Revisions

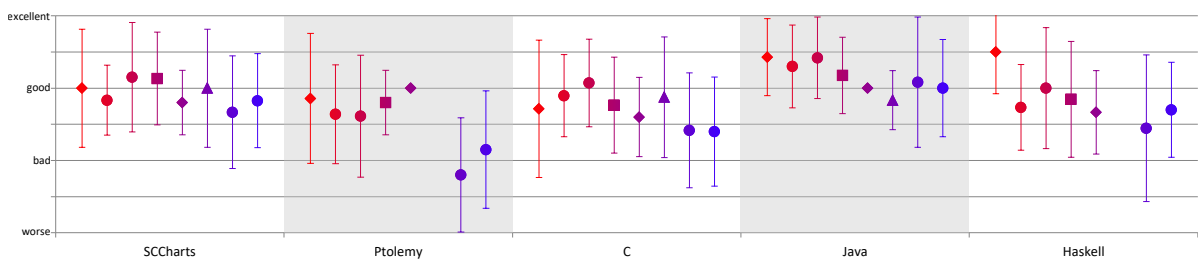


Figure 3.11.: Maintainability

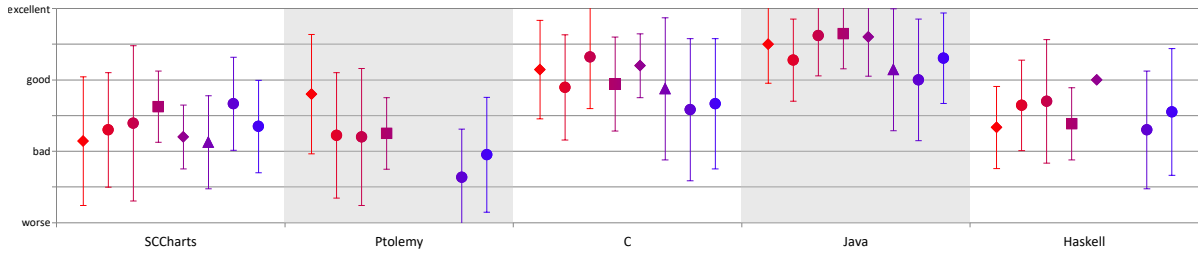


Figure 3.12.: Debugging

**Description** Fig. 3.11 shows the *maintainability* and Fig. 3.12 the *debugging* ratings. Both questions ask how difficult is it to change existing models/programs. The subtle difference here is that maintainability aims towards the enhancement and adaptation of an existing model and debugging tries to find and eliminate errors. While important for all programs, especially embedded systems tend to have long life cycles and maintenance becomes a crucial aspect over time. Furthermore, the synchronous MoCs are usually more restrictive than classical programming languages and causality errors may be hard to find and fix. Therefore, the developer requires more help from the IDE to solve these problems.

**Results** The maintainability ratings for SCCharts, also for the external group, are good and on par with C. While it still seems to be easier to maintain projects in classical programming languages, the ratings indicate that using a Statechart dialect is a viable solution w.r.t. maintainability. Furthermore, when different target codes or new code from legacy models are created, e.g., for upgraded hardware of legacy products, the model-based approach should be able to show its true maintenance potential. However, this feature was not covered by the tasks of the conducted studies and remains future work.

In comparison to classical programming, debugging is still the weak point of synchronous languages and SCCharts. The results are distinctly worse compared to C and Java. Causality problems, e.g., caused by cyclic dependencies, are often hard to spot during model creation time. Also, unsupported or broken features, such as array support for every new transformation, may not be as visible as it should be during modeling in such a large academic project. Depending on the project size, maintaining an overview is only possible with good module composition. During runtime, classical breakpoint debugging or assertions are not available in the actual SCCharts tooling.

To address these issues, recent SCCharts research focused on tools for finding issues, such as dedicated views (see Sec. 1.2.2) and debugging [Gri16]. However, even with simulation debuggers, which is also available in Ptolemy, new means to test and debug models during simulation and also during live execution should be developed. While the interactive incremental compiler allows some level of back-tracing during compilation and the overall tooling improved over the last four years, further work needs to be done w.r.t. debugging. Especially new users should be able to spot and fix issues easily and should not be burdened with maintaining a permanent overview over every facet of

the complex synchronous MoCs. Further ideas on how the model-based approach could improve on this subject will be discussed in Sec. 5.2.

### 3.3. Feature Aspects

This survey category discusses the importance of SCCharts language features. For each feature, the survey participants rated how important they deemed that feature w.r.t. relevance for their project. As a recall, in comparison to the study conducted in 2014 the project tasks and sizes differed. In 2014, the railway project required features that facilitate one large project connected to an existing C interface, whereas the NXT projects consisted out of several small models that interacted with the Mindstorms interface via the template engine (see Sec. 1.2.2). To structure the features, the aspects are grouped in five blocks, which results are each display in their own figure.

#### 3.3.1. Basic Transition Features

Fig. 3.13 shows essential transition features such as priorities, triggers, and effects.

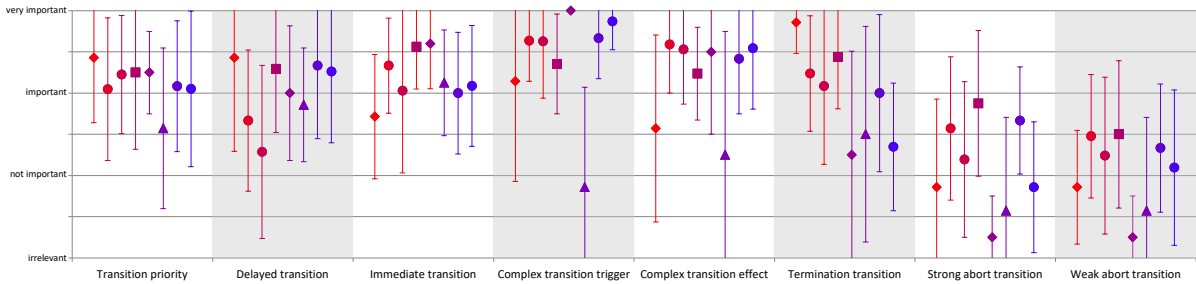


Figure 3.13.: Essential transition features

**Delay behavior** In the Mindstorms projects, delayed transitions played a less important role and immediate transitions were used more often. Especially the more complex Mindstorm tasks (see Chap. 2) often required instantaneous calculations, such as PID controller results, or decision making, e. g., via decisions trees. There is no state required and hence, consumption of time can be avoided. These tasks often did not require complex concurrency, and therefore avoided tick consumptions that might had been necessary because of causal relationships. In the future, we want SCCharts to give the modeler the opportunity to model in a more hybrid-like manner. Instantaneous calculations are modeled naturally in dataflow, whereas states take advantage of the Statechart’s way. The reduced amount of long immediate transition chains that emulate dataflow calculations seems unnatural and should not be necessary in the future.

**Trigger and effects** Especially when working with large models and dealing with many effects, sequences of transition effects make the model less readable. As can be seen in Figure 3.13, the railway team used fewer transition effects than the Mindstorm teams. They compensated the lack of effects on the transitions with entry actions in states (Sec. 3.3.2). This results in semantically equal behavior, but displays the models in a more compact way. As mentioned in Sec. 1.2.2, modern pragmatic features, such as label management, can also help to keep the model size reasonable.

**Preemption** To avoid scheduling issues in the railway projects, superstates were left via normal terminations in most cases. Preemption was used rarely. Since then, the abort transformations were improved. Preemption was used more often in the Mindstorm projects. Better overview due to the smaller project size might also contribute to this decision. This strengthens the argument that dedicated preemption mechanisms in the SCCharts kernel language are not necessary. However, it is still an open question if such kernel additions would be beneficial to ease the downstream compilation and the readability of the generated code.

### 3.3.2. History, Suspension, and Actions

Fig. 3.14 shows the participants' results regarding history and action features.

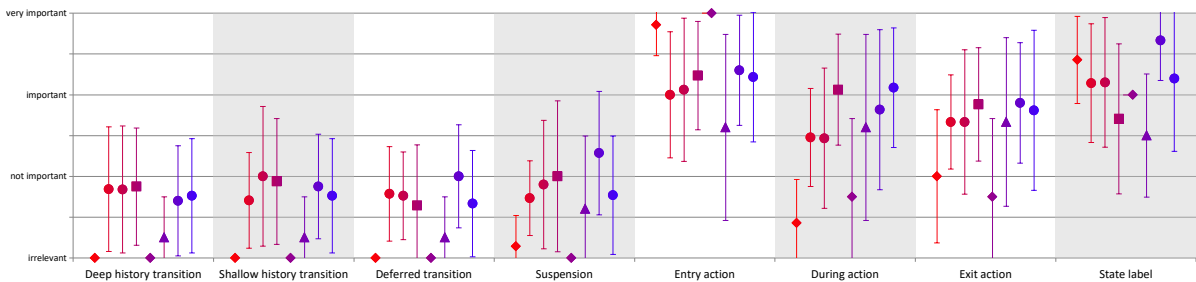


Figure 3.14.: History and local action features

**History** All history-related transition features were not deemed important for the given tasks. However, they are popular in other model-based languages, such as SyncCharts and Ptolemy. They have a moderate internal complexity in SCCharts (see Sec. 1.2) and could be excluded from the compilation chain to simplify the compilation when it comes to similar tasks.

**Suspension** Equal to the history ratings, suspension features were not required. It is noteworthy that suspension is part of Esterel's kernel language [PBEB07], but not deemed necessary, at least for the given tasks, in SCCharts.

**Actions** As mentioned before in Sec. 3.3.1, since the beginning of the use of SCCharts in teaching, users tend to write actions (especially entry actions) to model effects, mainly to keep the diagram sizes reasonable. This trend continued, even so the need might not be as urgent for the Mindstorms projects due to the smaller model sizes.

More recent projects made more use of during and exit actions. Again, smaller project sizes, the tasks themselves, and compiler improvements, which nowadays support concurrent actions better, make the use of these features more viable.

It is an interesting question if this way of modeling should be adapted as the main way of effect emission. In this case, the transitions themselves would only comprise triggers and the semantic duality of these approaches could be abolished.

### 3.3.3. Concurrency, Declarations, and Data Types

Fig. 3.15 shows the participants' results regarding concurrency, declarations, and different data types.

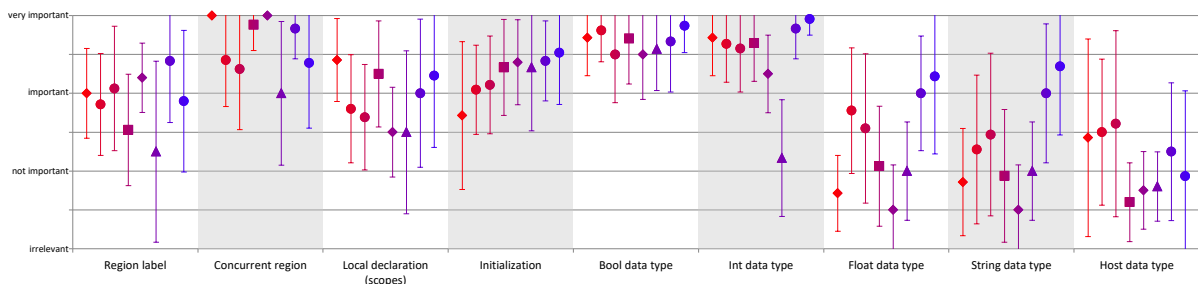


Figure 3.15.: Concurrency, declarations, and data types

**Concurrency** As one decisive feature of Synchronous Languages, concurrency was rated very important. While it may not be of paramount importance compared to the railway projects where eleven trains must perform the same tasks concurrently, there are also a multitude of jobs that can be solved concurrently in the Mindstorms' settings. After all, deterministic concurrency is one of the defining pillars that justify the existence of Synchronous Languages.

**Declaration** Scoping, together with local variables, and the possibility to initialize these variables were also rated important. These convenient structuring mechanisms come at low cost for the compiler and help to keep the project maintainable.

**Data Types** While boolean and integer data types were necessary from day one, floating point and string data types gained popularity over the course of time. The particular data type usage is use-case specific, but especially calculation-heavy program parts, such as PID controllers, often need fractions. In the Mindstorms setting, the string type is handy to give immediate feedback, because the Mindstorm unit has a display. This

is not the case in the railway setup, where string support is only needed for logging operations. New wrapper and deployment capabilities in the KIELER compiler (see also Sec. 1.2.2) also made the host data type less mandatory over time.

### 3.3.4. Additional Extended Features

Fig. 3.16 shows the participants' results for additional extended features, such as *Count Delay*, *Signals*, and *Referenced SCCharts*.

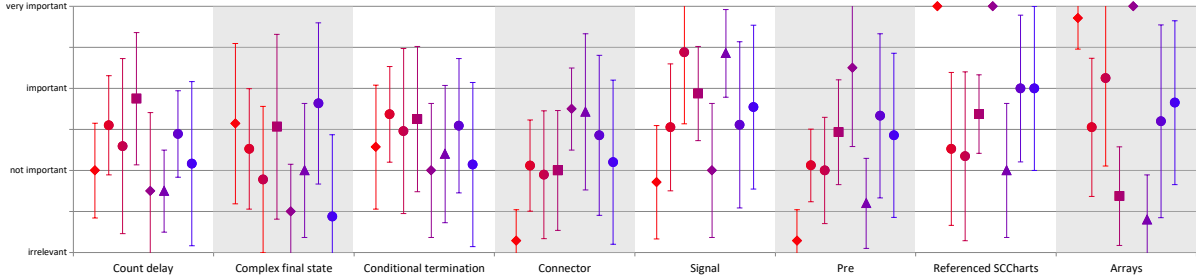


Figure 3.16.: Additional extended features

**Count Delay** An arguably important feature in synchronous languages in general is *count delay*. A count delay with an integer  $n$  delays a transition firing until the trigger condition evaluated to true  $n$  times. Though it was always a part of the SCCharts language set, the handling, especially when working with several count delay incarnations with the same signal, e. g., in concurrent context, sometimes becomes complicated. Depending on the actual model, this can lead to circular dependencies, which are hidden due to the extended nature of the feature. New scheduling possibilities [SSRvH19] and transformations that solve these problems are now part of KIELER SCCharts (see also Fig. 1.5). While it is sometimes handy to just state a count delay, e. g., to count seconds, for the given tasks, the count delay rating was below important.

**Complex Final State** In contrast to *simple final states*, which terminate a region as soon as they are reached, *complex final states* are final states that have inner behavior or can be left again. The region is only terminated if all concurrent regions of the parent superstate reach a final state. This is not a core feature of SCCharts and the extended transformation has a high internal complexity (see Sec. 1.2), which makes it not viable for the Mindstorms due to the resource limitations as discussed in Sec. 1.1.

**Conditional Termination** A *conditional termination* can only fire, so that the originating superstate is left, if its trigger evaluates to true. While this feature can be handy, it was not deemed mandatory.

**Connector** Although thought to be useful for describing certain flows in documentation, the *connector* feature did not seem to have any practical relevance in the projects. However, it is usually used to increase the readability and hence, the maintainability of the diagram. It also comes at low cost for the compiler.

**Signal** A *signal* is another prominent and mandatory feature of many synchronous languages. Usually, signals are necessary for concurrent communication. However, they are not mandatory in SCCharts as they can be fully emulated with boolean variables and the scheduling regime of SCCharts. This explains the high rating of the external student group, whose lecture focused on classical synchronous languages and not SCCharts-specific characteristics. Nonetheless, also the other groups find signals with their intrinsic absence semantics useful in SCCharts.

The emulation of signals in SCCharts introduces extra concurrent regions with implicit during actions, which complicate the compilation especially in combination with the abort transformation. They were used more often in the smaller projects and avoided in the railway projects. Implemented optimizations might have contributed to the fact that they are more widely used in later projects.

**Pre** Indispensable in classical synchronous languages, *pre* is not as important in SCCharts. While this might also depend on the particular tasks (the external group also rated *pre* unimportant), SCCharts mainly uses variables, which store their value even across tick boundaries. Furthermore, as *pre* is an extended feature in SCCharts with a moderate internal complexity (see Sec. 1.2), it can lead to resource issues in the context of Mindstorms (see Sec. 1.1).

**Referenced SCCharts** *Referenced SCCharts* are SCCharts' main mechanism to support modularity. SCCharts can reference other SCCharts models and include their behavior. While still important for structuring (see also the results for modularity in Sec. 3.2.5), the Mindstorms tasks' project sizes would also allow for models without referenced SCCharts, whereas the feature made the railway project practical in the first place. This is a confirmation that modularity is always good, but mandatory for larger projects.

**Arrays** The specific use-case also dictates if arrays are mandatory or not. The C API and especially the need to address many peripheral devices in the railway projects needed arrays to work efficiently. This was not necessary when working with the Mindstorms. Furthermore, the template engine (Sec. 1.2.2) makes direct host interaction obsolete.

### 3.3.5. Future Features

In the later surveys, we asked the participants, what possible future features, which were not or not fully implemented at the time of the participation, they deemed important. Fig. 3.17 shows the participants' results. Overall the participants' results in this category seem to be slightly undecided. However, we still evaluate the answers relative to each

other within one survey group to see which features were rated more important than others.

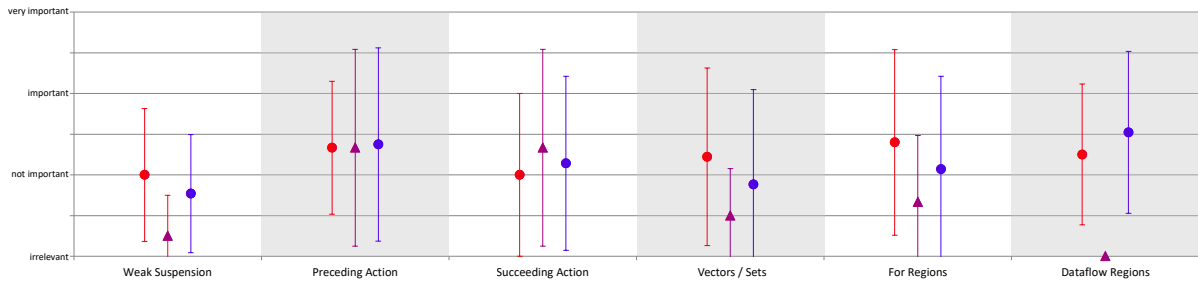


Figure 3.17.: Future features

**Weak Suspension** A *weak suspend* is implemented in languages such as Esterel V7 and in a limited version in Quartz. It allows the actual tick to finish, but restarts the current tick anew in the next tick. This feature is regarded highly situational. Furthermore, as the simple suspend is rarely used in the given tasks, most likely this feature is not needed in SCCharts.

**Additional Actions** Like SyncCharts, SCCharts handle entry actions and exit actions asymmetrically w.r.t. preemption. To give the modeler the possibility to decide whether or not an action should be preemptable, two new actions could be added. The participants' ratings are indecisive, but with a high standard deviation.

**Vectors / Sets** Vector assignments should ease the usability of array and large data structures. However, the low ratings and recent imperative coding style developments, e.g. imperative loops, (see Sec. 5.2), might deem this addition unnecessary.

**For Regions** For regions can duplicate region behavior, either for a specified range or an array. This convenient feature assigns an iterator variable to each duplicated region. While not critical, this feature saves modeling time and can increase readability and maintainability.

**Dataflow Regions** *Dataflow regions* allow direct computations within a region. This enables the modeler to create hybrid SCCharts and reduces the need to model immediate transition chains for computations without real state change. Calculation parts can be modeled in dataflow regions, whereas state-based parts use the traditional state-transition way.



### 3.4. Tooling Aspects

In the next set of questions in the survey (Appendix A) the participants should rate the overall quality of the SCCharts tools in the actual KIELER implementation. The tooling category also includes the results of the synchronous survey group  $\blacksquare$  from the Synchron Workshop 2016 in Bamberg. Their shortened survey can be found in Appendix B.



Figure 3.18.: Tool quality

**Overall Quality of SCCharts Tools** Fig. 3.18 compares the overall quality of the tools at the beginning of the projects with the end of the projects. The SCCharts tools were completely new when the first railway project began. They performed bad in larger projects, especially because there was no matured concept for modularity. As the tools improved and with the addition of the *referenced SCCharts* feature [SMSR<sup>+</sup>15], the rating increased significantly.

The Mindstorms projects did not suffer from these issues at the start of the projects. Moreover, they greatly benefit from the lessons learned during the railway project. Of course, the participants of the later projects did not have this perspective. However, even comparatively small changes in the usability are visible in the figure as the final rating also improved marginally during the department internal projects. Overall the SCCharts tooling is rated between **ok** and **advanced** at the end of all projects. While we are satisfied with the way this big academic product matured, there is still room for improvements.

#### 3.4.1. Model Creation and Debugging

When it comes to model creation and debugging, Fig. 3.19 shows that the discrepancy between small and large models was rated more extreme in the railway projects. Naturally, what was seen as a large model differs within the project groups. However, all participants agreed that debugging, particularly of large models, is hard. As already established in Section 3.2.6, debugging remains the Achilles' heel of SCCharts. However, the relative trends indicate that the tooling improved over time and we plan to focus even more on usability, maintainability, and debugging features.

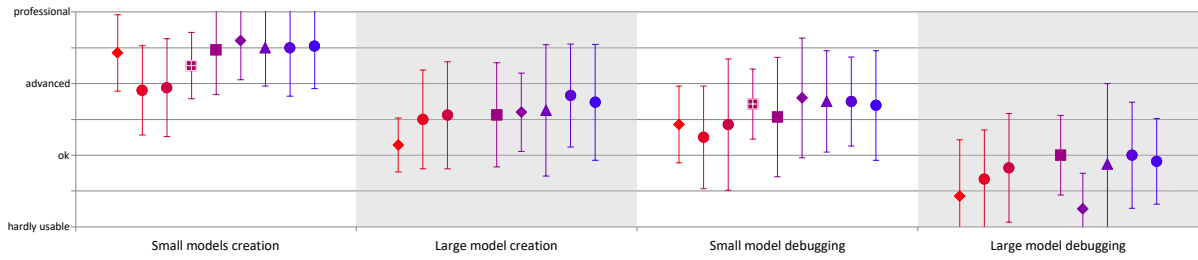


Figure 3.19.: Model creation & debugging

### 3.4.2. Further Tooling Aspects

Fig. 3.20 shows several aspects of the tooling. It is noteworthy that the ratings from the professional group are generally higher than the ratings of the student groups. This can be contributed to the fact that the professional participants have more experience in working with similar problem tasks and comparable tools. They also may have a deeper understanding about the issues that need to be solved and about the available alternatives.

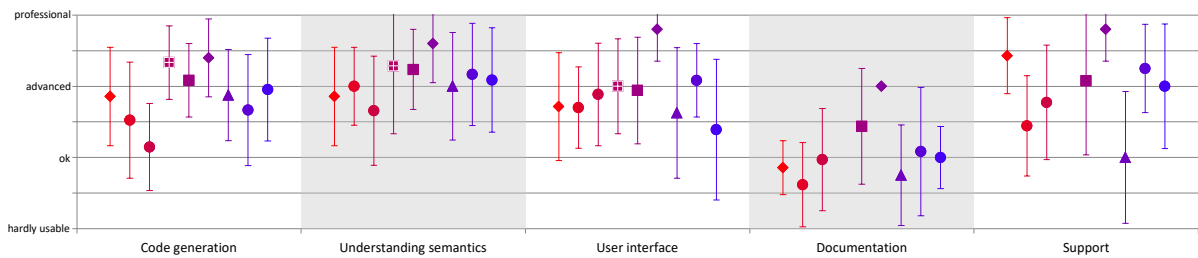


Figure 3.20.: Tooling aspects

**Code Generation** The code generation was rated worse in the first two Mindstorms projects, which can be attributed to the resource limitations discussed in Sec. 1.2.2. In the beginning, SCCharts were quite generous when it came to the usage of guard variables. Especially larger models, such as the Barcode Reader (see Sec. 2.4), resulted in larger programs that were problematic for the used leJOS firmware (see Sec. 1.1). In later versions, several optimizations [SSRvH18d, Bus16] were added to make larger models executable on leJOS. Overall, the code generation was deemed around the **advanced** mark. The optimization of the generated code remains an open topic, especially when working with limited resource systems.

**Understanding Semantics** Clear understanding of the semantics of language features is important. Since all ratings are above average, the tooling provides clear representations of the features in use and the processes involved during model creation and compilation are comprehensible.

**User Interface** The rating of the user interface nearly stayed the same in all projects. There is a distinct drop in the ratings in the last study group. A reason for this might be that some students in this lecture run did not complete the initial leJOS tutorial mentioned in Chap. 2. This resulted in problems later on since they faced errors as a result of their missing leJOS or Java installation. Another reason for it might be the recent popularity of other IDEs, such as VSCode<sup>3</sup> and IntelliJ<sup>4</sup>, which use different, arguably novel, UI concepts than the Eclipse-based framework. Active research [Dom18, Ren18] in this area examines these pragmatic questions.

**Documentation** Although the documentation improved over time, this is still another weak spot of the SCCharts project. Documentation and examples are present, and got expanded especially before the last iteration of the embedded systems class. However, the latest improvements do not seem to have a big impact on the ratings and are not enough to give a better than **ok** impression. More extensive documentation and/or better ways of presenting the actual state of the project should be explored in the future.

**Support** Naturally, the in-house projects scored better in the support ratings. The latest improvements (see Sec. 1.2.2), such as the template engine and more people working on more topics simultaneously in the KIELER project, increased these ratings again over time.

---

<sup>3</sup><https://code.visualstudio.com>

<sup>4</sup><https://www.jetbrains.com/idea>

## 4. Related Work

There exist some popular programming language rankings, such as the rankings from RedMonk<sup>1</sup> or the TIOBE Index<sup>2</sup>. A brief summary of criteria, which can be considered when ranking programming languages, can be found in the notes<sup>3</sup> of Prof. Toal. Many of the technical criteria can be found in the SCCharts surveys. However, the goal of the SCCharts surveys was to make sure that the language and the tooling is able to compete with other mainstream languages, particularly in the context of embedded systems and teaching, and not to replace any established paradigms.

Furthermore, SCCharts was used in several other academic projects, which will be discussed in the following sections.

### 4.1. Railway Projects

#### 4.1.1. Railway Demonstrator

The model railway demonstrator<sup>4</sup> shown in Figure 4.1 is a practical lab at Kiel University since 1995. It was originally built and run by the group of Prof. Kluge and in 2006 transferred to the real-time and embedded systems group<sup>5</sup> of Prof. Reinhard von Hanxle-

<sup>1</sup><https://redmonk.com>

<sup>2</sup><https://www.tiobe.com/tiobe-index>

<sup>3</sup><https://cs.lmu.edu/~ray/notes/evaluatingprogramminglanguages/>

<sup>4</sup><http://www.informatik.uni-kiel.de/~railway>

<sup>5</sup><http://www.rtsys.informatik.uni-kiel.de>

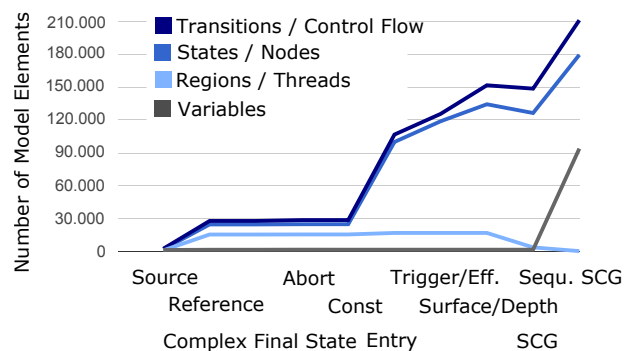


Figure 4.1.: Railway installation (left) and first SCCharts railway controller (right) (from [Mot17])

den. Stephan Hörmann developed the third generation [Höh06] of the demonstrator in 2006 which was operational until April 2015. The 4th generation of the model railway was developed by Nis Wechselberg [Wec15] based on a conceptual proposal [Mot14] and is in place since April 2015.

The track layout of the demonstrator was inspired by a mountain pass in Canada, the *Kicking Horse Pass*<sup>6</sup>. Since its initial version, the model railway installation has the following characteristics: Scale H0, approximately 130 meter of tracks, 48 blocks of track segments, 28 switch points, 56 signal lights, and 80 reed contacts.

### 4.1.2. Train Controlling

Over the years, the railway installation has been subject to numerous significant changes in its track layout details and the interaction with a controller software.

The latest version connects all hardware periphery to numerous Arduino nodes which are USB connected to a small number of Raspberry Pi micro computers. These are connected via Ethernet to a central controller computer which runs a C program interpreting the reed contact information and controlling the track segments, switch points, and signal lights. Based on this scenario, in various practical labs the task was given to model a controller program (in various synchronous languages, such as SCADE, Esterel, Ptolemy, and SCCharts) and generate code for the railway installation.

### 4.1.3. Practical Labs

In summer term 2014 our group hosted and supervised a railway project<sup>7</sup> in order to evaluate SCCharts as a language and our KIELER SCCharts tooling for the first time using the railway installation. During the railway project seven participants worked about 6 months with SCCharts building a controller that runs up to 11 concurrent trains on the installation. Detailed information about the project and its results can be discovered separately in a technical report [SMSR<sup>+</sup>15].

During this lab, an SCCharts controller was modeled which fully expands to 135,000 states, 152,000 transitions, and 17,000 concurrent regions after eliminating all reference states by a reference state compiler transformation. 1,628 states were modeled manually together with 2,219 transition and 183 concurrent regions. The eleven train model railway controller had a final size of roughly 400,000 lines of C code.

Figure 4.1 shows the measurements of the number of model elements for the SCCharts model railway controller example at every intermediate stage of the compile chain during expansion. The result and suggests how much complexity of the model could be hidden by using Extended SCCharts features for modeling the complex behavior of this controller

The students were not only using our SCCharts compiler tool chain, but also struggling with teething troubles of our early prototype compiler. This resulted in much improvements of the stability and quality of the compiler.

<sup>6</sup>[http://en.wikipedia.org/wiki/Kicking\\_Horse\\_Pass](http://en.wikipedia.org/wiki/Kicking_Horse_Pass)

<sup>7</sup><http://rtsys.informatik.uni-kiel.de/confluence/display/SS14Railway>

Other practical labs with similar tasks had been conducted in the following years, further improving the SCCharts language (features) and the compiler and overall tooling.

#### 4.1.4. Survey

After the project, the students were asked about their experience with SCCharts and the KIELER SCCharts tooling. The results are documented in a technical report [SMSR<sup>+</sup>15]. It is noteworthy that the number of seven participants for this first survey were quite small. However, as this report shows, following surveys, also in other scenarios such as Lego Mindstorms, confirmed most of the initial collected numbers.

## 4.2. Teaching Synchronous Languages

The KIELER SCCharts tool is also used for teaching synchronous languages at Kiel University. In the lecture, the theoretical concepts of the synchronous MoC as well as practical experience in synchronous programming is taught. Alongside SCCharts, the synchronous languages Esterel [Ber00] and Lustre [HCRP91] are introduced and covered in practical exercises. In contrast to the Embedded Real-Time lectures, the focus lies more on the theoretical concepts applied in synchronous languages, rather than the broader topic of embedded systems and has no involvement of special hardware, such as LEGO Mindstorms. For example, one task is to model a controller for a backhoe loader, displayed in Fig. 1.6, where the KIELER tool provides the simulation and visualization to solve the task. The task itself was initially developed for Esterel<sup>8</sup> by Timothy Bourke. It is also available as an example for Zélus<sup>9</sup> which is a Lustre-like synchronous language with Ordinary Differential Equations (ODEs) [BP13].

### 4.2.1. Survey

The synchronous languages lecture also included surveys to evaluate the SCCharts languages, especially in comparison to other synchronous languages. For the sake of completeness, the results are included in the evaluation in Chap. 3 and in general reflect the trends presented here. Regarding the language features, SCCharts includes many advanced language constructs initially introduced for Esterel and Lustre. Hence, students who are more familiar with their semantics, due to the different thematic focus in the lecture, may use and rate them differently than those from the Embedded Real-Time lectures.

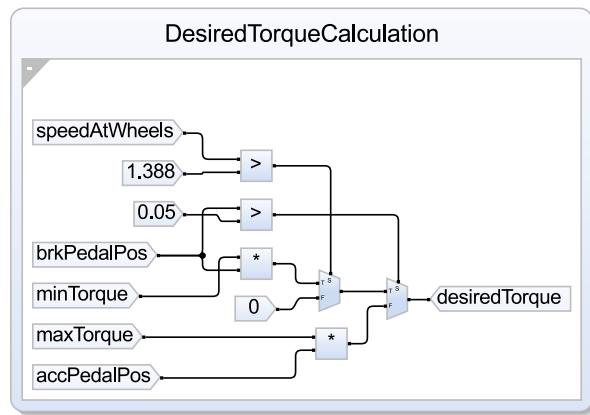
---

<sup>8</sup><https://www.tbrk.org/esterel/backhoe.html>

<sup>9</sup><http://zelus.di.ens.fr/examples.html#ex-backhoe>



(a) One of the validation tracks made in IPG Carmaker



(b) SCCharts dataflow model of the desired torque calculation

Figure 4.2.: Using SCCharts Models in Simulink to Model an Electronic Control Unit [SSSRvH19]

### 4.3. Raceyard

SCCharts have been successfully used to model a controller for the Kieler Formula Student Team Raceyard [SSSRvH19]. The Formula Student is an international design competition for students, where every year the aim for each team is to design, construct, and test a race car. These race cars are then compared and judged during official events. Since 2005 Team Raceyard takes part in the Formula Student as the official team of the University of Applied Sciences Kiel.

For the task of designing and testing the control systems of the Electric Control Unit (ECU), Raceyard uses the MathWorks' modeling software Simulink<sup>10</sup>. Of special importance for Raceyard are the built-in blocks for PID controllers and the provision of scopes and displays to view every signal inside the system during and after runtime. Verifying as well as fine-tuning the ECU is done via the simulation software IPG Carmaker<sup>11</sup>. IPG Carmaker provides a virtual 3D environment and simulated drivers of varying abilities so that the effects of the ECU can intuitively be seen on the 3D car model. Due to integration of IPG Carmaker into Simulink, values from the simulation such as the velocity of the car or the forces acting upon it can be measured and saved using the previously described scope- and display-blocks in Simulink. Following the test-phase, the Simulink model is then converted into C code and put on an STM32F40 micro controller inside the car.

This work shows how a functionally equivalent system can be designed by utilizing the visual synchronous language SCCharts in the academic open-source project KIELER. A complete controller model was modeled in KIELER and validated to behave the same

<sup>10</sup><https://mathworks.com/products/simulink.html>

<sup>11</sup><https://ipg-automotive.com>

as the original controller both in Simulink directly as well as in the 3D simulation environment IPG Carmaker as can be seen in Fig. 4.2a. Tests on the performance of both controllers show that while a slowdown can be observed when comparing the generated C Code, simulation time in IPG Carmaker only increases by a negligible factor. Also, the newly developed dataflow extension was tested in this project, shown in Fig. 4.2b, as dataflow modeling becomes the natural choice when designing computation-heavy controllers, such as PIDs.



## 5. Wrap-Up

### 5.1. Conclusion

The SCCharts surveys confirm that SCCharts and the tooling can compete with other mainstream languages, particularly in the context of embedded systems and teaching. In many aspects SCCharts performs as good or even better than other synchronous languages. Overall, the participants perceived SCCharts as understandable, simplistic, and maintainable. In the previous report [SMSR<sup>+</sup>15], debugging, composability, and team development were rated as under-performing. While the whole framework improved significantly since then, particularly debugging complex causality problems remain an issue, especially as the model sizes increase. The development environment must provide the modeler with adequate tools to combat this difficulty. Therefore, easier scheduling paradigms [SSRvH19, SRSM19], better tooling measures, such as specialized views [SSRvH18b, WSRSvH18], and human-traceable code generation approaches [SMvH18] are active research areas. Also performance issues with respect to large model browsing should be addressed pragmatically. Dedicated filters, such as a *Google Maps* like approach and sophisticated label management [Sch19], can help to keep large models manageable.

### 5.2. Future Work

Besides promoting a more hybrid-like modeling paradigm, as already sketched out in Sec. 4.3 and also investigated elsewhere [Uml15, Gri19], there are several active research topics w.r.t. SCCharts. Towards the languages development, novel object-oriented modeling practices are discussed in Sec. 5.2.1. Sec. 5.2.2 sketches out recent model checking capabilities regarding the SCCharts tooling.

#### 5.2.1. Object Orientation

SCCharts are recently extended towards object-oriented modeling [SRSM19]. This includes support for inheritance in developing SCCharts modules that allows to express commonalities and specialize behavior by overriding. Additionally methods, are introduced to defines reusable, imperative, and state-less code sections. This allows to express algorithmic problems, that may not be expressed as easily in a state machine or in dataflow. In methods SCCharts also supports loops, which have quickly proven themselves adjuvant in handling arrays. Fig. 5.1 illustrates the inheritance relationship between two SCCharts. These SCCharts also contain methods, visualized by gray regions that

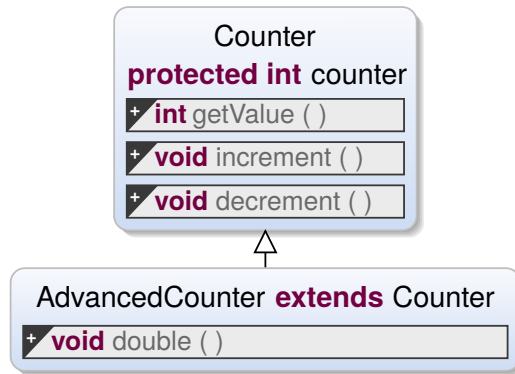


Figure 5.1.: Inheritance and method regions in SCCharts [SRSM19]

have a name, return type, and in this example empty parameters. Furthermore, the type system was also extended to provide classes for encapsulating multiple variables and their methods.

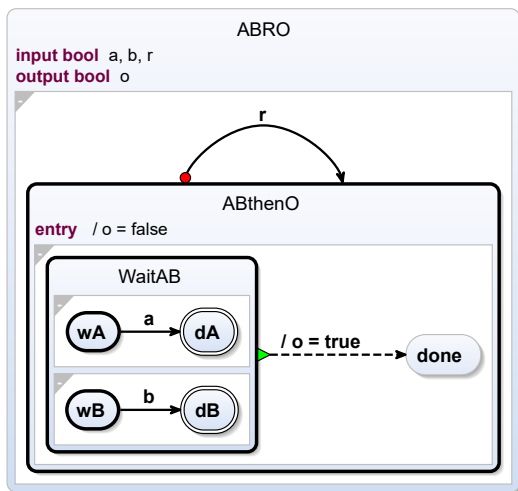
All these feature are about to be evaluated in future surveys, to measure their viability and effect on the usability of SCCharts.

## 5.2.2. Model Checking

Fig. 5.2a shows the ABRO model, the “Hello World!” of Synchronous Languages, which illustrates concurrency, hierarchy and strong aborts. The model waits concurrently for the inputs **a** and **b**. When both have been received, the output **o** is set to true. The input signal **r** resets the model, such that **o** is set to false and the model waits for **a** and **b** again. Since **r** triggers a strong abort, the simultaneous arrival of **a**, **b**, and **r** does not emit an **o**, because the inner behavior of **ABthenO** is preempted strongly.

Model checking properties can be defined for this behavior and checked automatically in the newest version of the KIELER SCCharts tools. Fig. 5.2b shows two examples: The invariant states that **o** cannot be present if **r** is present. The LTL property, (for Linear Temporal Logic), says that in any tick (G) if the next tick (X) has **a**, **b**, and not **r**, then **o** will be set.

Model checking should be promoted further in KIELER as it strengthen the argument for using modeling languages further. It can be also used to teach model checking principals and temporal logic. The interested reader can explore more about model checking for SCCharts elsewhere [Sta19].



(a) ABRO – The synchronous “Hello World!”

- Invariant property:  $r \rightarrow !o$
- LTL property:  $G X ((a \ \& \ b \ \& \ !r) \rightarrow o)$

(b) Model checking properties

Figure 5.2.: Model Checking for SCCharts [Sta19]

# Bibliography

- [Ber00] Gérard Berry. The foundations of Esterel. In *Proof, Language, and Interaction: Essays in Honour of Robin Milner*, pages 425–454, Cambridge, MA, USA, 2000. MIT Press.
- [BP13] Timothy Bourke and Marc Pouzet. Zélus: a synchronous language with odes. In *Proceedings of the 16th international conference on Hybrid systems: computation and control, HSCC 2013*, pages 113–118, Philadelphia, PA, USA, April 2013.
- [Bus16] Jonas Busse. SCCharts modeling for embedded systems with limited resources. Bachelor thesis, Kiel University, Department of Computer Science, September 2016. <https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/jbus-bt.pdf>.
- [Dom18] Sören Domrös. Moving model-driven engineering from Eclipse to web technologies. Master’s thesis, Kiel University, Department of Computer Science, November 2018. <https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/sdo-mt.pdf>.
- [Gri16] Lena Grimm. Debugging SCCharts. Bachelor’s thesis, Kiel University, Department of Computer Science, September 2016. <https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/lgr-bt.pdf>.
- [Gri19] Lena Grimm. From Lustre to graphical dataflow programs. Master’s thesis, Kiel University, Department of Computer Science, May 2019. <https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/lgr-mt.pdf>.
- [HCRP91] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, Sept. 1991.
- [Höh06] Stephan Höhrmann. Entwicklung eines modularen Feldbussystems zur Steuerung einer Modellbahnanlage. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, March 2006. <https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/sho-dt.pdf>.
- [Mot14] Christian Motika. Model Railway 2.0 - An Exploration of Alternatives for Interface Hardware, 2014.
- [Mot17] Christian Motika. *SCCharts—Language and Interactive Incremental Implementation*. Number 2017/2 in Kiel Computer Science Series. Department of Computer Science, 2017. Dissertation, Faculty of Engineering, Christian-Albrechts-Universität zu Kiel.
- [PBEB07] Dumitru Potop-Butucaru, Stephen A. Edwards, and Gérard Berry. *Compiling Esterel*. Springer, May 2007.

- [Ren18] Niklas Rentz. Moving transient views from Eclipse to web technologies. Master’s thesis, Kiel University, Department of Computer Science, November 2018. <https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/nir-mt.pdf>.
- [Sch19] Christoph Daniel Schulze. *Text in Diagrams—Challenges to and Opportunities of Automatic Layout*. Number 2019/1 in Kiel Computer Science Series. Department of Computer Science, 2019. Dissertation, Faculty of Engineering, Christian-Albrechts-Universität zu Kiel.
- [SMSR<sup>+</sup>15] Steven Smyth, Christian Motika, Alexander Schulz-Rosengarten, Nis Boerge Wechselberg, Carsten Sprung, and Reinhard von Hanxleden. SCCharts: the railway project report. Technical Report 1510, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, August 2015. ISSN 2192-6247.
- [SMvH18] Steven Smyth, Christian Motika, and Reinhard von Hanxleden. Synthesizing manually verifiable code for statecharts. In *Proc. Reactive and Event-based Languages & Systems (REBLS ’18), Workshop at the ACM SIGPLAN conference on Systems, Programming, Languages and Applications: Software for Humanity (SPLASH)*, Boston, MA, USA, November 2018.
- [SRSM19] Alexander Schulz-Rosengarten, Steven Smyth, and Michael Mendler. Towards object-oriented modeling in SCCharts. In *Proc. Forum on Specification and Design Languages (FDL ’19)*, Southampton, UK, September 2019.
- [SSRvH18a] Steven Smyth, Alexander Schulz-Rosengarten, and Reinhard von Hanxleden. Guidance in model-based compilations. Poster presented at the Doctoral Symposium of the 8th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2018), Limassol, Cyprus, November 2018.
- [SSRvH18b] Steven Smyth, Alexander Schulz-Rosengarten, and Reinhard von Hanxleden. Guidance in model-based compilations. In *Proceedings of the 8th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA ’18), Doctoral Symposium*, Electronic Communications of the EASST, Limassol, Cyprus, November 2018. in press.
- [SSRvH18c] Steven Smyth, Alexander Schulz-Rosengarten, and Reinhard von Hanxleden. Towards interactive compilation models. In *Proceedings of the 8th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2018)*, volume 11244 of *LNCS*, pages 246–260, Limassol, Cyprus, November 2018. Springer.
- [SSRvH18d] Steven Smyth, Alexander Schulz-Rosengarten, and Reinhard von Hanxleden. Watch your compiler work — Compiler models and environments. Technical Report 1806, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, July 2018. ISSN 2192-6247.
- [SSRvH19] Steven Smyth, Alexander Schulz-Rosengarten, and Reinhard von Hanxleden. Practical causality handling for synchronous languages. In *Proc. Design, Automation and Test in Europe Conference (DATE ’19)*, Florence, Italy, March 2019. IEEE.

- [SSSRvH19] Monty Santarossa, Steven Smyth, Alexander Schulz-Rosengarten, and Reinhard von Hanxleden. Using SCCharts models in Simulink to model an electric control unit. Technical Report 1903, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, July 2019. ISSN 2192-6247.
- [Sta19] Andreas Stange. Model checking for SCCharts. Master’s thesis, Kiel University, Department of Computer Science, May 2019. <https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/aas-mt.pdf>.
- [Uml15] Axel Umland. Konzept zur Erweiterung von SCCharts um Datenfluss. Diploma thesis, Kiel University, Department of Computer Science, March 2015. <https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/aum-dt.pdf>.
- [vHDM<sup>+</sup>14] Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer, and Owen O’Brien. SCCharts: Sequentially Constructive Statecharts for safety-critical applications. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI ’14)*, pages 372–383, Edinburgh, UK, June 2014. ACM.
- [Wec15] Nis B. Wechselberg. Model railway 4.0. Master thesis, Kiel University, Department of Computer Science, March 2015. <https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/nbw-mt.pdf>.
- [WSRSvH18] Nis Wechselberg, Alexander Schulz-Rosengarten, Steven Smyth, and Reinhard von Hanxleden. Augmenting state models with data flow. In Marten Lohstroh, Patricia Derler, and Marjan Sirjani, editors, *Principles of Modeling: Essays Dedicated to Edward A. Lee on the Occasion of His 60th Birthday*, LNCS, pages 504–523. Springer International Publishing, 2018.

## A. SCCharts Survey

Appendix A contains the survey that was handed out to the participants of the projects. To compare the results while working on the SCCharts compiler and tool chain, it is recommended to also use this survey in upcoming SCCharts projects. The following list describes the version history of the survey.

**Version 1.0** Initial version

**Version 1.1** Added new features *referenced SCCharts* and *arrays*

**Version 1.1s** Simplified version, contains only the most important questions

**Version 1.2s** Simplified version, but includes a layout question

**Version 1.3** Online version of 1.2s, including new languages features *vectors* and *for regions*

**Version 1.3.1** Same as 1.3 with new language feature *dataflow regions*

# SCCHARTS SURVEY FORM

Rev. 1.2s

Project Name: \_\_\_\_\_ Size: \_\_\_\_\_ (# of involved people)

Project Start/End: \_\_\_\_\_ Date: \_\_\_\_\_

Start Time: \_\_\_\_\_ (current time when starting this survey)

## I) About yourself

1) In which semester are you (in your current degree program)?

2) What degree do you pursue?  Bachelor

 Master

3) Which **modeling tools** have you used before?

---

---

---

---

4) Which **synchronous/dataflow languages** have you used before?

---

---

---

---



## II) Project involvement

---

5a) Estimate distribution of work for **your part** of the project

# of SCCharts <b>modeled</b>	<input type="text"/>
# of SCCharts <b>reviewed</b>	<input type="text"/>
# of SCCharts <b>documented</b>	<input type="text"/>
# of SCCharts <b>tested</b>	<input type="text"/>
# of <b>additional SCCharts-generating scripts</b> written	<input type="text"/>

5b) Of 100% time you spent in the project, you ...

% <b>modeled</b> with SCCharts	<input type="text"/>
% written in <b>host language</b>	<input type="text"/>
% written <b>additional scripts</b>	<input type="text"/>
% elaborated <b>documentation</b>	<input type="text"/>
% did <b>testing</b>	<input type="text"/>
% <b>discussion time</b>	<input type="text"/>
% other: <input type="text"/>	<input type="text"/>
% other: <input type="text"/>	<input type="text"/>

6) Estimate your SCCharts skills **before** this project

<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
<b>professional</b>	<b>advanced</b>	<b>ok</b>	<b>greenhorn</b>
know all features and can argue for or against them, could teach others	know most features, have some feelings for/against usage	know main features and can implement requested functionality	know some features and can read most models, implement running (small) models

7) Estimate your SCCharts skills **after** this project

<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
<b>professional</b>	<b>advanced</b>	<b>ok</b>	<b>greenhorn</b>
know all features and can argue for or against them, could teach others	know most features, have some feelings for/against usage	know main features and can implement requested functionality	know some features and can read most models, implement running (small) models

8) To what extent you think you could make use of dataflow (DF) for this project? (If you don't know about DF, you may skip this question)

<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
<b>perfect</b>	<b>most parts</b>	<b>some parts</b>	<b>nothing</b>
everything should be modeled with data flow	mostly dataflow should be used, only some parts need control flow	mostly control flow should be used, only some parts need data flow	data flow seems not the right thing to be used at all

9) To which extent would you like to use the following modeling/programming languages **for this project?**

**SCADE**

**perfect** use for everything

**good** use for most parts

**Ok**  
use for up to 50% of the models

things can be **hardly** modeled with this language

**don't know** this language

**Esterel**

**perfect** use for everything

**good** use for most parts

**Ok**  
use for up to 50% of the models

things can be **hardly** modeled with this language

**don't know** this language

**SyncCharts**

**perfect** use for everything

**good** use for most parts

**Ok**  
use for up to 50% of the models

things can be **hardly** modeled with this language

**don't know** this language

**SCCharts**

**perfect** use for everything

**good** use for most parts

**Ok**  
use for up to 50% of the models

things can be **hardly** modeled with this language

**don't know** this language

**Ptolemy**

**perfect** use for everything

**good** use for most parts

**Ok**  
use for up to 50% of the models

things can be **hardly** modeled with this language

**don't know** this language

**C**

**perfect** use for everything

**good** use for most parts

**Ok**  
use for up to 50% of the models

things can be **hardly** modeled with this language

**don't know** this language

**Java**

**perfect** use for everything

**good** use for most parts

**Ok**  
use for up to 50% of the models

things can be **hardly** modeled with this language

**don't know** this language

**Haskell**

**perfect** use for everything

**good** use for most parts

**Ok**  
use for up to 50% of the models

things can be **hardly** modeled with this language

**don't know** this language

### III) Language aspects

---

10) Grade the following languages regarding the handling of each aspect. Use ++(excellent), +(good), -(bad), --(worse) as marks. You may skip a grade if you are uncertain. Comments about your choices are appreciated:

Problem/Aspect	SCADE	Esterel	SyncCharts	SCCharts	Ptolemy	C	Java	Haskel
<b>General determinism</b> <small>Achieve deterministic behavior</small>								
<b>Deterministic concurrency</b> <small>Avoid race conditions</small>								
<b>Sequentiality</b> <small>Express sequential parts</small>								
<b>Composability</b> <small>Compose sub-solutions to an overall solution</small>								
<b>Solving abstract problems</b>								
<b>Solving low-level problems</b>								
<b>Understandability</b> <small>Overview of large projects</small>								
<b>Simplicity</b> <small>Learning curve</small>								
<b>Separate Timing &amp; Functionality</b>								



12) Your opinion: Rate the following features of SCCharts w.r.t. their significance **for the project**:

(use the grading scheme from 9: ++(very important), +(important), -(not important),-- (irrelevant) or blank)

SCCharts Feature	Grade
Transition priority	
Delayed transition	
Immediate transition	
Complex transition trigger (e.g., (A & B)   (C & D))	
Complex transition effect (e.g., A = B&C; D = E)	
Termination transition & Final States	
Strong abort transition	
Weak abort transition	
Deep History transition	
Shallow History transition	
Deferred transition	
Suspension	
Entry action	
During action	
Exit action	
State label	
Region label	
Concurrent region	
Local declaration (scopes)	
Initialization	
Bool data type	
Int data type	
Float data type	
String data type	
Host data type	
Count delay	
Complex final state (e.g., with outgoing transitions)	
Conditional termination	
Connector	
Signal	
Pre-Operator	
Referenced SCCharts	
Arrays	

13) What were the **most challenging functionalities** to be implemented in the Project? Why?

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

14) Which extended features did you miss? Describe the features and explain why.

---

---

---

---

---

---

---

## IV) Tooling aspects

---

15) Your opinion about the **overall quality** of the SCCharts development tools you worked with, compared to other modeling/programming environments:

At the **beginning** of the project:

<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
<b>Professional</b> as other mature open source/commercial products	<b>Advanced</b> as other smaller commercial or open source products	<b>Ok</b> As beta versions of commercial software or Freeware	<b>Hardly usable</b> like alpha versions or private/obsolete projects

At the **end** of the project:

<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
<b>Professional</b> as other mature open source/commercial products	<b>Advanced</b> as other smaller commercial or open source products	<b>Ok</b> As beta versions of commercial software or Freeware	<b>Hardly usable</b> like alpha versions or private/obsolete projects

16) For **specific use cases**, rate the quality of the SCCharts development tools you worked with, compared to other modeling/programming environments (at the end of the project), write down possible enhancements:

Creation/**modeling of small** models:

<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
<b>Professional</b> as other mature open source/commercial products	<b>Advanced</b> as other smaller commercial or open source products	<b>Ok</b> As beta versions of commercial software or Freeware	<b>Hardly usable</b> like alpha versions or private/obsolete projects

Enhancements:

---



## Creation/modeling of large models:



**Professional** as other mature open source/commercial products



**Advanced** as other smaller commercial or open source products



**Ok**  
As beta versions of commercial software or Freeware



**Hardly usable** like alpha versions or private/obsolete projects

---

Enhancements:

## Debugging of small models:



**Professional** as other mature open source/commercial products



**Advanced** as other smaller commercial or open source products



**Ok**  
As beta versions of commercial software or Freeware



**Hardly usable** like alpha versions or private/obsolete projects

---

Enhancements:

## Debugging of large models:



**Professional** as other mature open source/commercial products



**Advanced** as other smaller commercial or open source products



**Ok**  
As beta versions of commercial software or Freeware



**Hardly usable** like alpha versions or private/obsolete projects

---

Enhancements:

## Code generation:



**Professional** as other mature open source/commercial products



**Advanced** as other smaller commercial or open source products



**Ok**  
As beta versions of commercial software or Freeware



**Hardly usable** like alpha versions or private/obsolete projects

---

Enhancements:

## Understanding the language semantics:



**Professional** as other mature open source/commercial products



**Advanced** as other smaller commercial or open source products



**Ok**  
As beta versions of commercial software or Freeware



**Hardly usable** like alpha versions or private/obsolete projects

---

Enhancements:

## User Interface:



**Professional** as other mature open source/commercial products



**Advanced** as other smaller commercial or open source products



**Ok**  
As beta versions of commercial software or Freeware



**Hardly usable** like alpha versions or private/obsolete projects

---

Enhancements:

## Documentation:



**Professional** as other mature open source/commercial products



**Advanced** as other smaller commercial or open source products



**Ok**  
As beta versions of commercial software or Freeware



**Hardly usable** like alpha versions or private/obsolete projects

---

Enhancements:

## Support:



**Professional** as other mature open source/commercial products



**Advanced** as other smaller commercial or open source products



**Ok**  
As beta versions of commercial software or Freeware



**Hardly usable** like alpha versions or private/obsolete projects

---

Enhancements:

17) Were any timing-related problems relevant?

Yes, execution time calculation for **whole program**

Was an execution time analysis for the whole program (several ticks) needed? (WCET)

Describe your solution and/or how the tooling could have helped:

---

Yes, execution time calculation for **one tick**

Was an execution time analysis for the one tick needed? (WCRT)

Describe your solution and/or how the tooling could have helped:

---

Yes, **synchronization** between ticks

Synchronizing concurrent threads at specific tick boundaries, e.g., "Is thread x at tick c in state s?" or "Who does what in which tick?"

Describe your solution and/or how the tooling could have helped:

---

Yes, namely: \_\_\_\_\_

Describe your solution and/or how the tooling could have helped:

---

No

18) Would the automatic display of (partial) execution time information, e.g., for regions, in the graphical diagram have been helpful?

Yes

No

19) Do you have any other remarks about the SCCharts tools/compiler you would like to share?

**Pro:**

---

---

---

---

---

---

---

---

---

---

**Contra:**

---

---

---

---

---

---

---

---

---

---

20) End Time: \_\_\_\_\_ (current time when finishing this survey)

21) Layout-Question

Did you use the Label-Management?

If yes then please describe when you used it and what options you preferred. Also you may want to give additional comments or suggestions

---

---

---

Did you have problems with long labels?

Describe how you solved these problems or dealt with them. Do you have any suggestions how the tooling should have helped you here?

---

---

---

Did you have problems with the layout in general?

Describe your problems, suggestions here. Did you use the KLayout or Graphviz? Why?

---

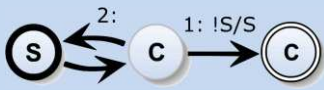
---

---

Thank you!

## **B. SCCharts Short Survey**

The short survey is restricted to one page and only includes tooling related questions. It was designed to be attached to a tutorial print-out for workshop sessions. The survey version number is **1.1t**.



# SCCHARTS SURVEY FORM

Have you use KIELER SCCharts **before**?

- Yes**  
Please let us know in the comments below, where you used it.
- No**

Can you image to use KIELER SCCharts in the future?

- Yes**     in class     in projects     for fun
- No**

Rate the quality of the SCCharts development tools you worked with.

Creation/**Modeling** of models:

- Professional**  
as other mature open source or commercial products
- Advanced**  
as other smaller commercial or open source products
- Ok**  
as beta version of commercial software or Freeware
- Hardly usable**  
like alpha versions or private/obsolete projects

**Debugging** of models:

- Professional**  
as other mature open source or commercial products
- Advanced**  
as other smaller commercial or open source products
- Ok**  
as beta version of commercial software or Freeware
- Hardly usable**  
like alpha versions or private/obsolete projects

**Interactive** compilation/code generation:

- Professional**  
as other mature open source or commercial products
- Advanced**  
as other smaller commercial or open source products
- Ok**  
as beta version of commercial software or Freeware
- Hardly usable**  
like alpha versions or private/obsolete projects

**Wrapper\Template** code generation:

- Professional**  
as other mature open source or commercial products
- Advanced**  
as other smaller commercial or open source products
- Ok**  
as beta version of commercial software or Freeware
- Hardly usable**  
like alpha versions or private/obsolete projects

**Understanding** the language semantics:

- Professional**  
as other mature open source or commercial products
- Advanced**  
as other smaller commercial or open source products
- Ok**  
as beta version of commercial software or Freeware
- Hardly usable**  
like alpha versions or private/obsolete projects

**User interface:**

- Professional**  
as other mature open source or commercial products
- Advanced**  
as other smaller commercial or open source products
- Ok**  
as beta version of commercial software or Freeware
- Hardly usable**  
like alpha versions or private/obsolete projects

**Comments** on KIELER SCCharts:

(E.g., what do you think is missing for the release of the KIELER SCCharts development tools?)

**Comments** on this tutorial:

(E.g., what can we do to improve it? Was it helpful? Did you enjoy it?)

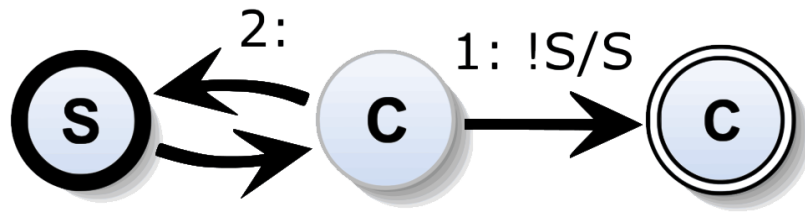
You can use the back page of this form to leave more detailed comments and/or suggestions.

**We appreciate your feedback! Thank you very much!**

## **C. SCCharts Tutorial**

The SCCharts Tutorial as it was presented during the Synchron Workshop in December 2016 in Bamberg.





# SCCharts Tutorial

SYNCHRON'16, Bamberg



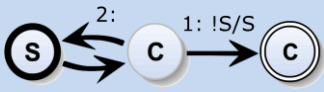
## KIELER SCCharts

- |   |    |
|---|----|
| 1. Tutorial Preparations                | 1  |
| 2. Exercise I: Textical Modeling        | 4  |
| 3. Exercise II: Interactive Compilation | 8  |
| 4. Exercise III: Simulation             | 13 |
| 5. Bonus Exercise IV: SCG               | 18 |

## Lego Mindstorms

- |   |    |
|---|----|
| 6. Lego Mindstorms Preparations             | 23 |
| 7. Exercise V: SCCharts for Lego Mindstorms | 28 |
| 8. Exercise VI: SCCharts Pathfinder         | 33 |
| 9. Questions & Feedback                     | 34 |





## 1 Tutorial Preparations

Welcome to the SCCharts Tutorial! To start, please copy the Eclipse-based KIELER SCCharts application from the provided USB thumb drive to your hard drive.

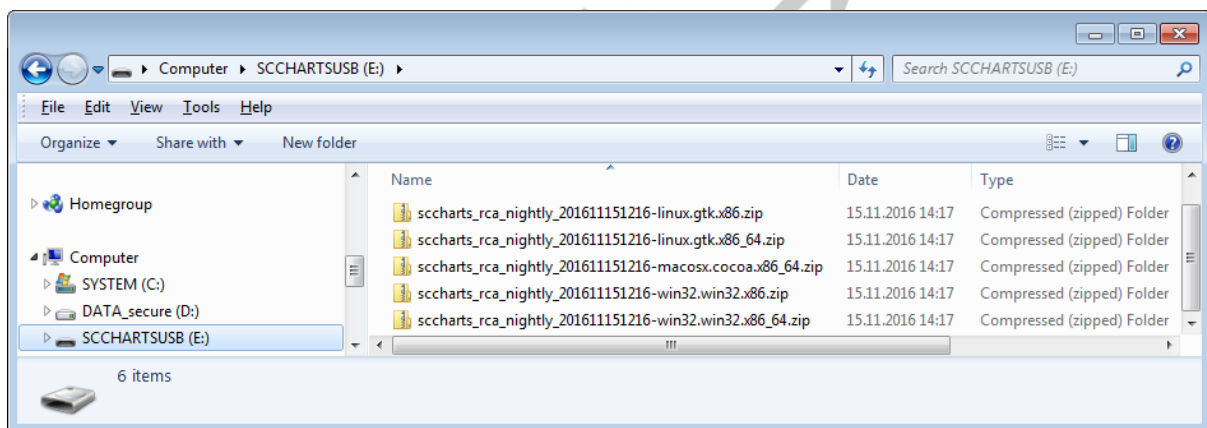
Additionally, you will need a **Java 8 SDK** on your machine. For Windows and the Lego tutorial part you will need the **32bit** version of Java and KIELER SCCharts. You will also find Java 8 installation files on the USB thumb drive in case you need to install it.

The following section, Section 1.1a, describes how to copy the appropriate files using a concrete example build. Note that the USB stick will contain the most recent build instead.

Alternatively, you read this tutorial documentation after the workshop and thus do not have a USB drive with the installation, then you can download the proper release candidate or the nightly RCA as explained in Section 1.1b.

### 1.1a Copy from USB Stick

- Plug in the provided USB stick: **/USBstick/KIELERSCCharts/**
- Choose a version suited for your OS and copy it to your computer.



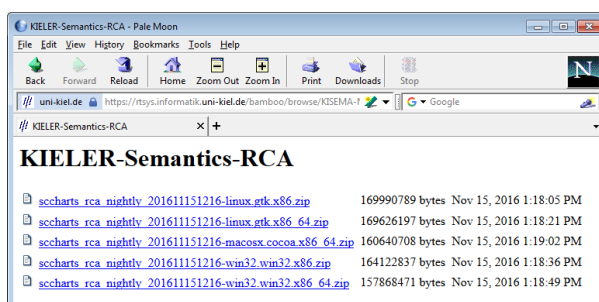
### 1.1b Download

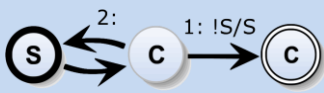
- Use the following URL:

[http://rtsys.informatik.uni-kiel.de/~kieler/files/release\\_sccharts\\_0.12.0/](http://rtsys.informatik.uni-kiel.de/~kieler/files/release_sccharts_0.12.0/)

or <http://tinyurl.com/scchartsrelease12>

- Choose a version suited for your OS and download it to your computer.





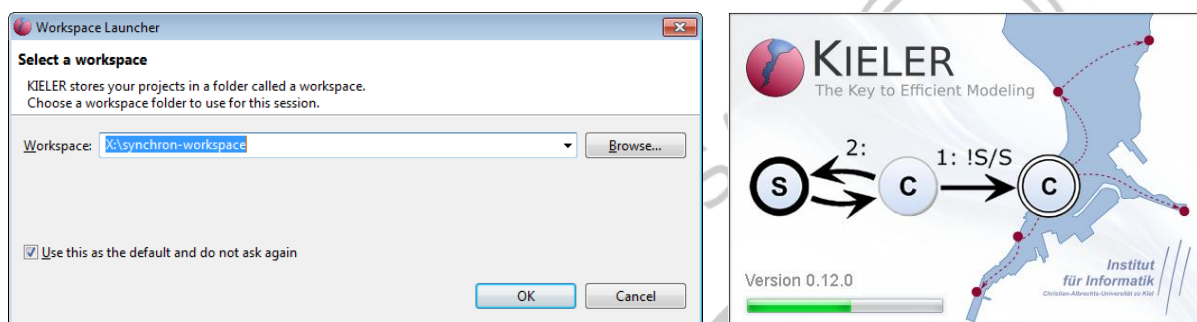
## 1.2 Extract / Unzip

Untar or unzip the KIELER SCCharts Tools in a folder of your harddrive.

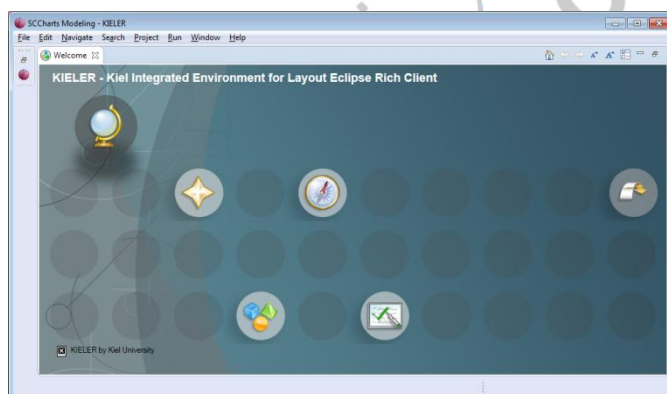
## 1.3 Start KIELER SCCharts Tools

After you have copied and extracted the KIELER SCCharts Tools, you can start the KIELER.exe or KIELER.app. For Mac, you may need to set your security settings such that non-appstore applications are allowed to be started.

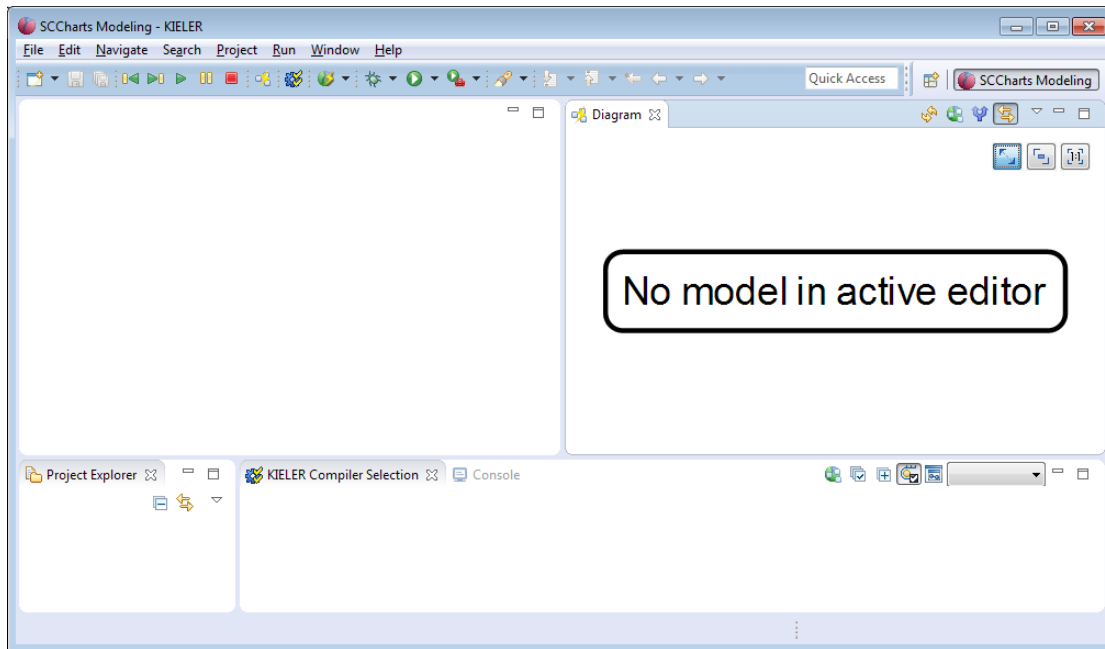
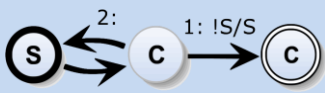
When you start the KIELER SCCharts Tools for the first time, choose an appropriate workspace path on your harddrive. You might later want to copy files there, so remember the path. For convenience, you may check the "[x] Use this as the default and do not ask again" option.



After that, KIELER should start and you should see the splash screen (see above). Then you should see the following welcome screen:

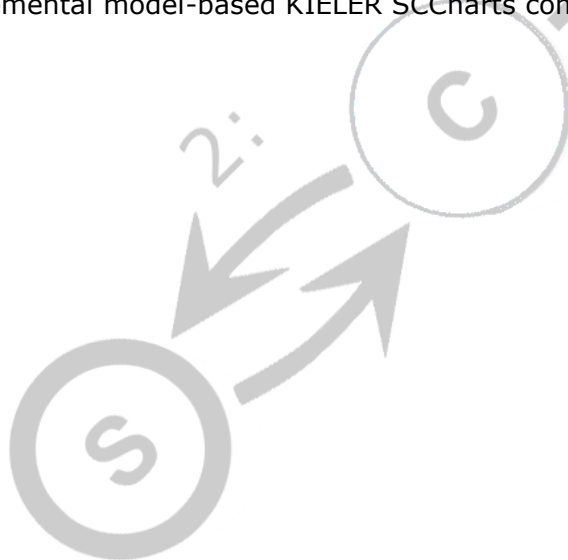


Close this tab. This will bring you to the KIELER SCCharts modeling perspective that should look similar to the following screenshot:



If your screen is missing the displayed *Diagram* and *KIELER Compiler Selection* view then you might not have the Java 8 SDK installed or Eclipse may use the wrong Java version if you have more than one installed on you system. Please use the provided Java 8 installation files on the USB stick.

Congratulations, you are now ready to model SCCharts and to get used to the interactive and incremental model-based KIELER SCCharts compiler. :-)

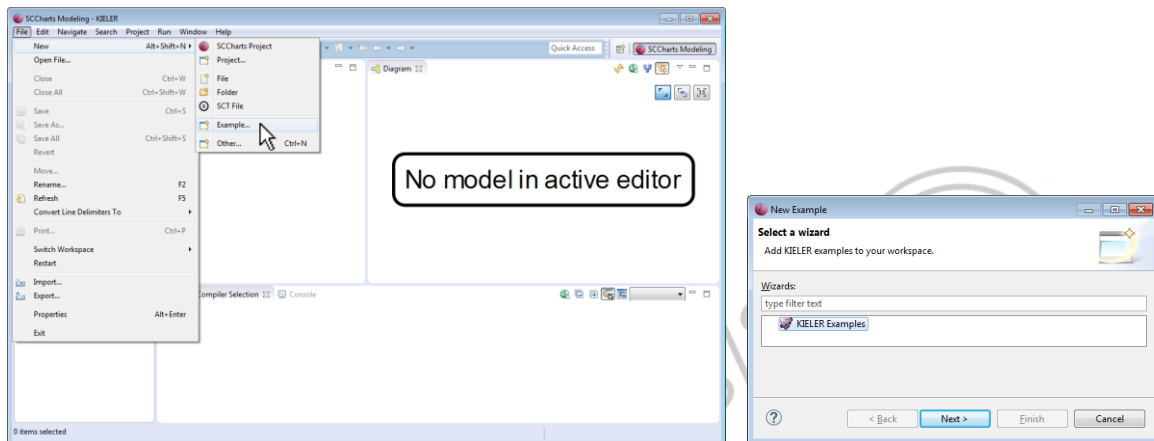


## 2. Exercise I: Textical Modeling

Learning Objective: Familiarize yourself with getting examples and textical SCCharts modeling.

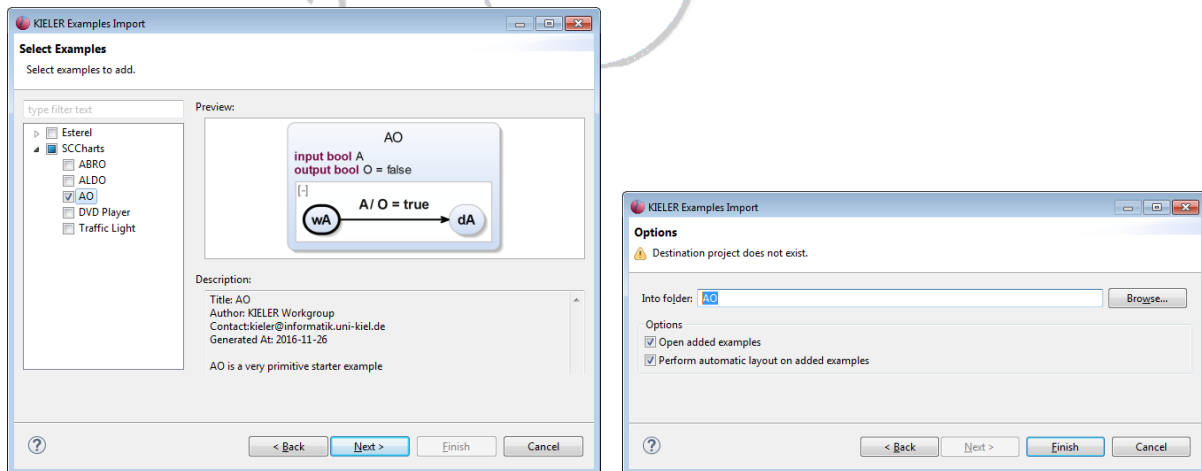
### 2.1 Getting SCCharts Examples

Click on "File" -> "New" -> "Example..." in the main menu. Then select "KIELER Examples" and continue with "Next >".

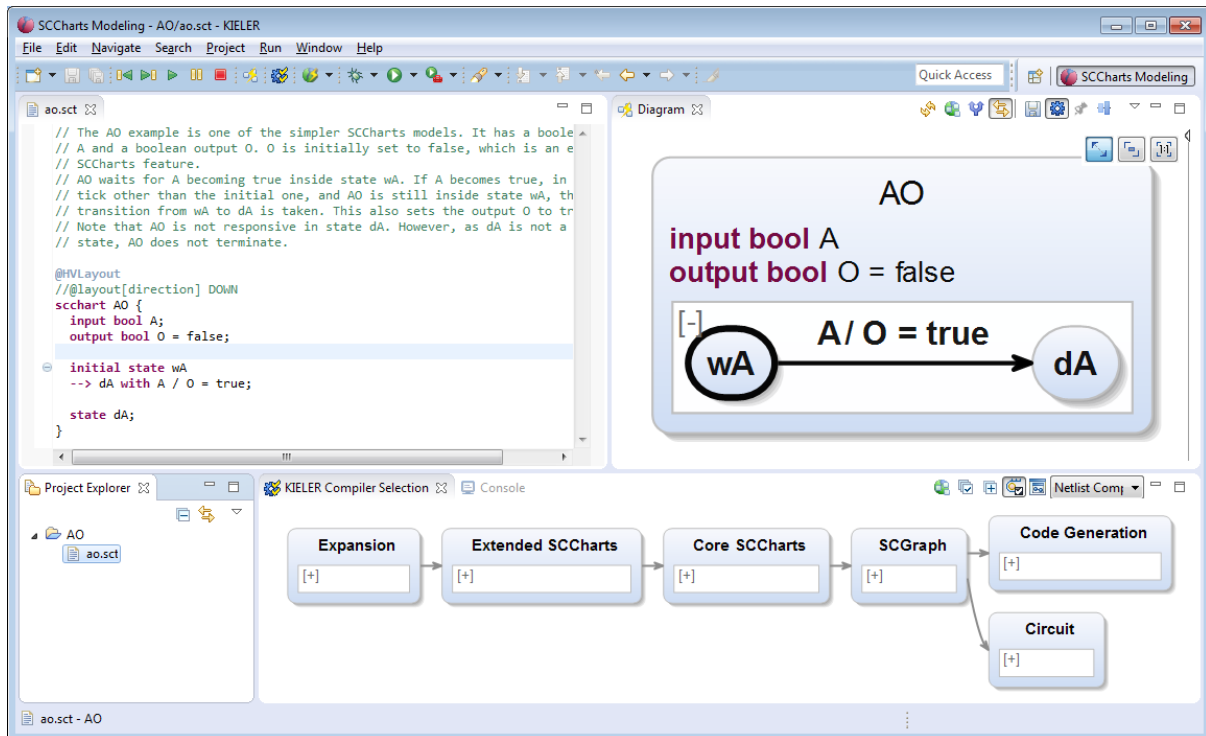


Now you may choose from various examples.

Choose the AO example by checking its checkbox and clicking "Next >". Then enter a proper name that will become the name of the Eclipse project, e.g., call it "AO".



After clicking "Finish", your screen should look similar to the following one. The textical SCCharts editor is opened on the left side, the automatically created diagram is shown on the right side, the Project Explorer is found on the lower left side and the KIELER Compiler Selection is visible in the lower right part of the window.



## 2.2 Textual Modeling Basics

You may now edit the AO example. Expect the diagram to be updated whenever you save your model (<Ctrl>+<S> or <Cmd>+<S>).

First, let's add another output variable O2 and initialize it to true. Just modify the code as follows:

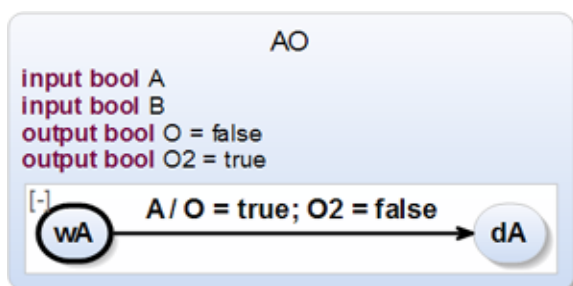
```

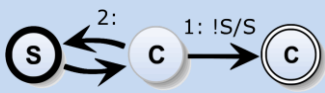
scchart AO {
  input bool A;
  input bool B;
  output bool O = false;
  output bool O2 = true;

  initial state wA
  --> dA with A / O = true; O2 = false;

  state dA;
}
  
```

After saving the model, the diagram will update accordingly and show the following graphical SCChart:





## 2.3 Textual SCCharts Editor

### Code Formatter

The editor is equipped with an automated code formatter that can be called using `<Ctrl>+<Shift>+<F>`. This can be useful any time when editing an SCChart. For example if you remove some line breaks, your code may look like as follows:

```
scchart AO {
  input bool A;
  output bool O = false; output bool O2 = true;
  initial state wA --> dA with A / O = true; O2 = false;
  state dA;
}
```

Now, after pressing `<Ctrl>+<Shift>+<F>`, the code formatter re-arranges the SCT code and makes it better readable. It arranges the interface part of a state at the top and separates it and all child states by a blank line. Outgoing transitions of a state are put at the bottom of a state declaration, after the internal behavior which is declared inside curly brackets "`{ ... }`", each transition in a new line.

```
scchart AO {
  input bool A;
  output bool O = false;
  output bool O2 = true;

  initial state wA
  --> dA with A / O = true; O2 = false;

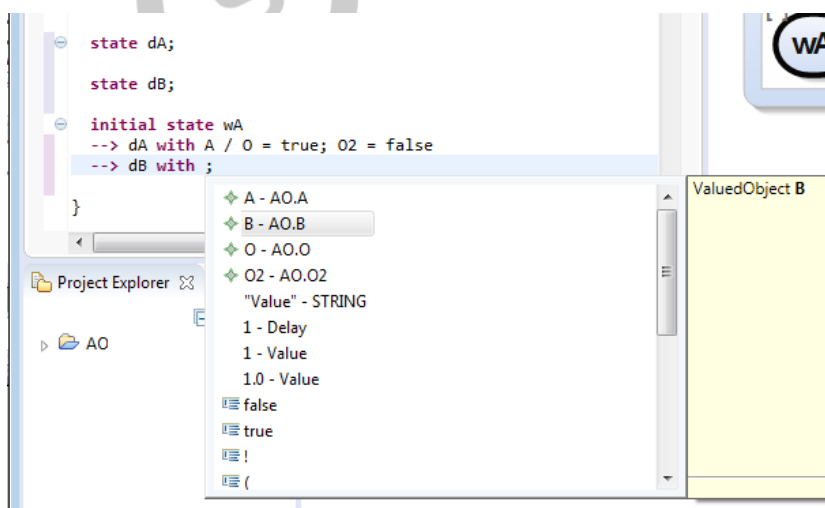
  state dA;
}
```

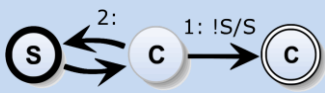
### Content Assist

The editor is equipped with a content assist feature which helps you writing SCT code more quickly. It also guides you towards syntactically correct SCT code if, for example you are not sure about the concrete order of specific keywords.

By pressing `<Ctrl>+<Space>` you activate the content assist.

Try to define a new bool input "B", a new state "dB", and add another transition from "wA" to "dB" in case of "B" becomes true. Set "O2" to false only in this transition.





## Syntax Validation

The editor is equipped with a syntax validator which checks the SCT model for syntactical problems or errors. It is called automatically while typing. You will notice possible error or warning markers beside the concerning lines.

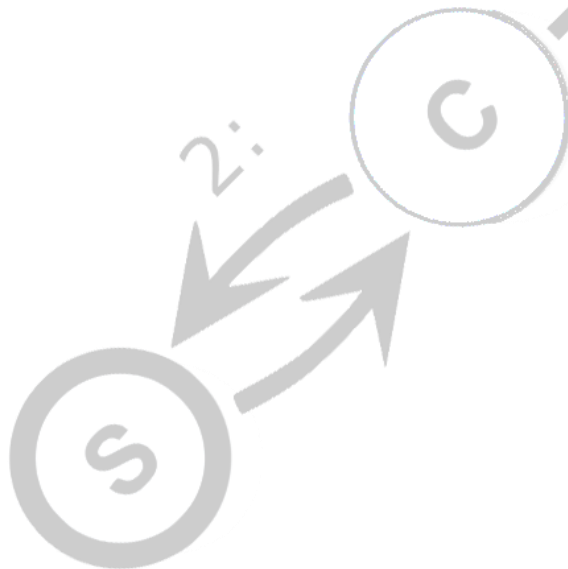
Try to remove the "initial" keyword from the state "wB".

```

scchart AO {
  input bool A;
  input bool B;
  output bool O = false;
  output bool O2 = true;

  state wA
  --> dA with A / O = true
  --> dB with B / O2 = false;
}
  
```

If you hover the mouse over these markers you get a hint what the problem is. In this case now all states become not reachable (which is a warning because this does not harm but in the end is dead code that will be eliminated). Additionally, as every region must have an initial state and the region in "AO" doesn't have one any more now, this will result in an error marker telling that there is no initial state found.





## 3. Exercise II: Interactive Compilation

Learning Objective: Familiarize yourself with the interactive incremental compilation tool chain for SCCharts.

### 3.1 Interactive Compilation

Start again with the AO example:

```

scchart AO {
  input bool A;
  output bool O = false;

  initial state wA
  --> dA with A / O = true;

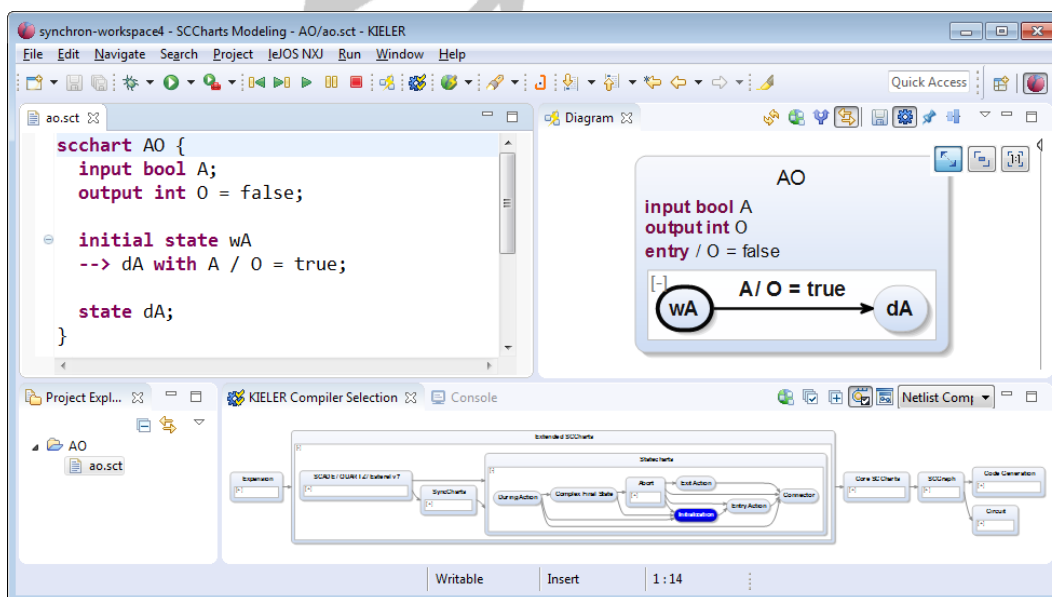
  state dA;
}

```

Note that the "O = false" is the use of the *initialization* feature. While compiling AO, initializations are replaced by entry actions which will perform the task of initializing a variable.

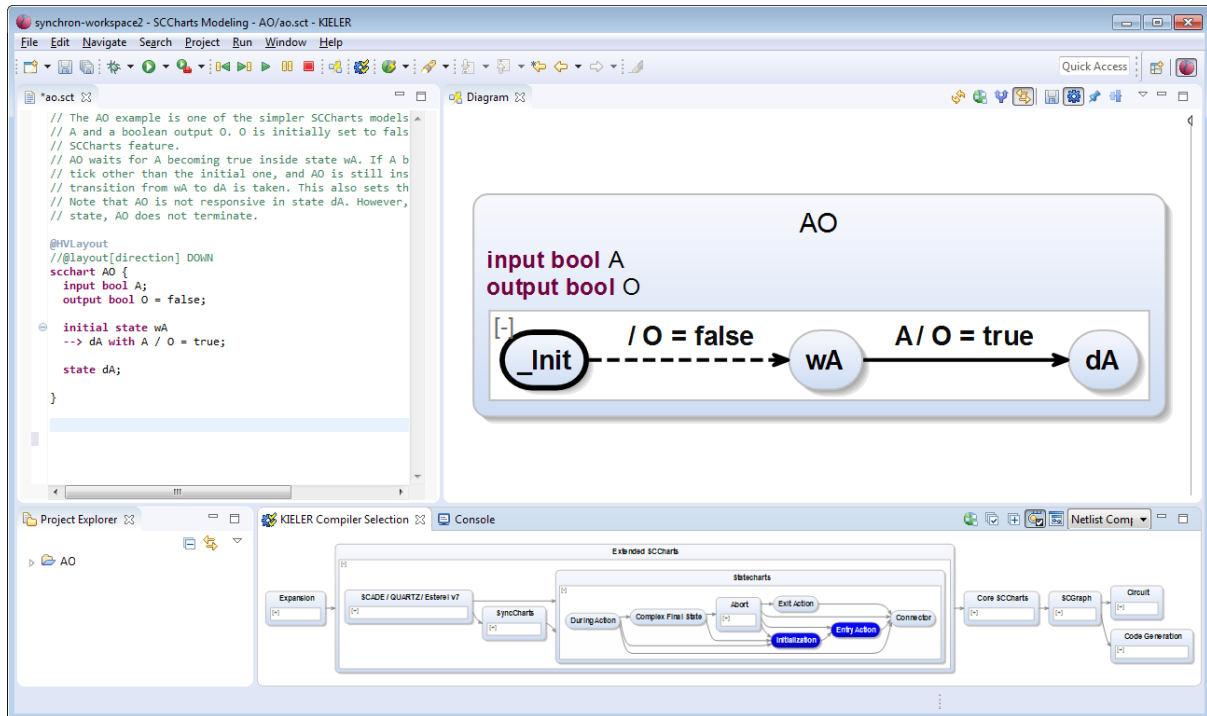
To inspect what this concretely means for your model, use the KIELER Compiler Selection view and select the "Initialization" feature transformation there. The model diagram will update and show the transformed model with entry actions as follows. Note that you need to first expand the "Extended SCCharts" feature group and then the "Statecharts" feature group in order to see the "Initialization" feature transformation.

Note that the diagram in the KIELER Compiler Selection re-uses the syntax of SCCharts but actually is not an SCChart. Nodes in this diagram represent feature transformations or feature groups and transitions represent dependencies. A dependency between two feature transformations simply means a certain order for applying these transformations.



For example, because the "Initialization" feature transformation produced new entry actions, it makes sense to transform the "Entry Action" feature only after the "Initialization" feature.

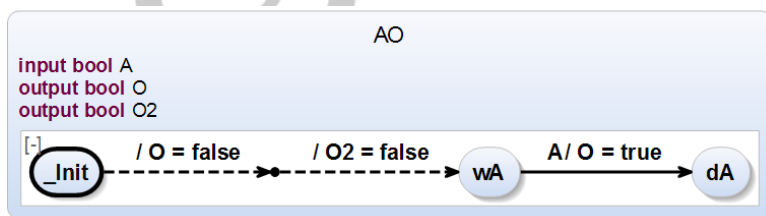
After also selecting the "Entry Action" feature transformation the transformed SCCharts looks like follows:



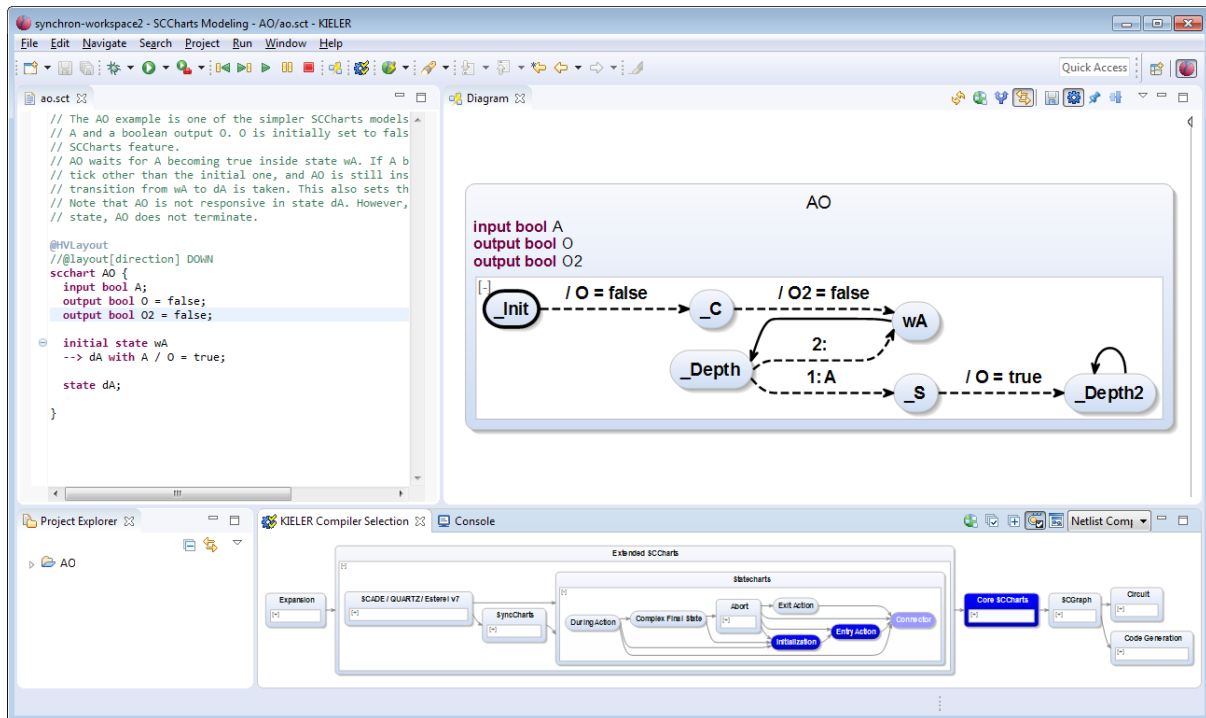
## 3.2 Interactive Compilation Modeling

Even if you select certain transformations to apply in the KIELER Compiler Selection view, you can still edit the original SCChart. The output will then be the updated original model with the selected feature transformations applied such that the selected features are not contained in the (intermediate) model any more. These feature are replaced by more basic features. For example "Entry" is a more basic feature compared to "Initialization".

You can add another interface declaration line, introducing another output variable O2 that is initialized to false. The intermediate transformed SCChart diagram should look like this:

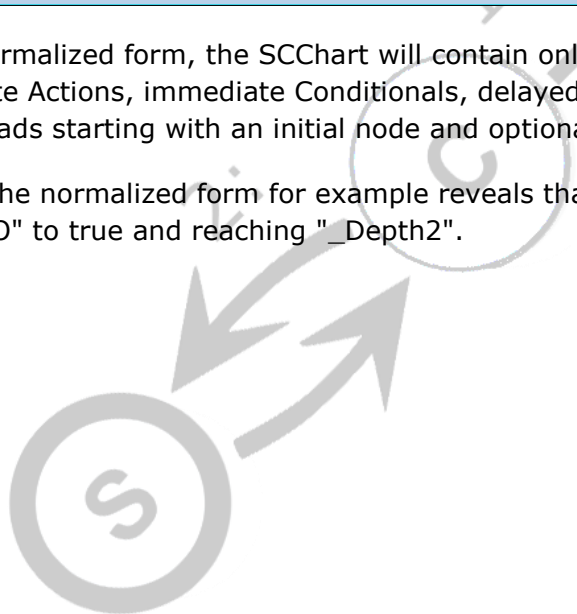


You could further experiment with transforming the intermediate result down to a Normalized Core SCChart by selecting all "Core SCCharts" feature transformations. The result should look similar to the following diagram:



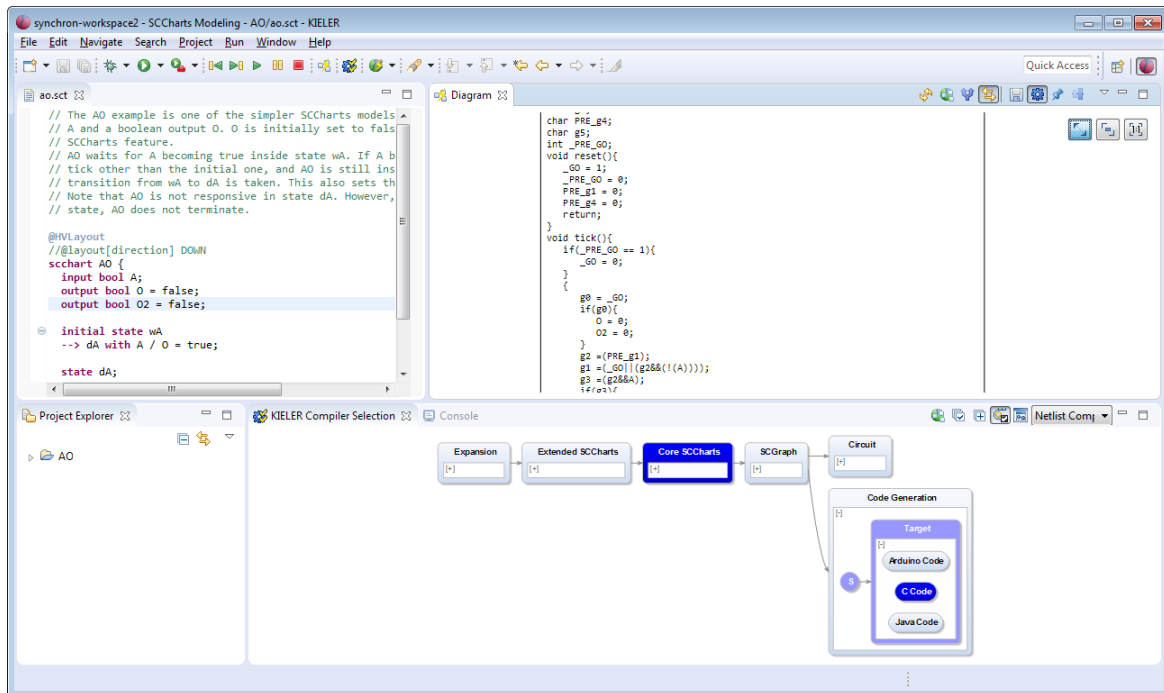
In the normalized form, the SCChart will contain only 5 primitive patterns which are immediate Actions, immediate Conditionals, delayed Pause, hierarchy with Fork/Join, and Threads starting with an initial node and optionally final states.

For AO, the normalized form for example reveals that the model never terminates after setting "O" to true and reaching "\_Depth2".



## 3.3 Generating Code

Code can be created easily by selecting a transformation from the KIELER Compiler Selection in the "Code Generation" feature group.



If you doubleclick the code preview of the Diagram view, the code will open in a textual editor.

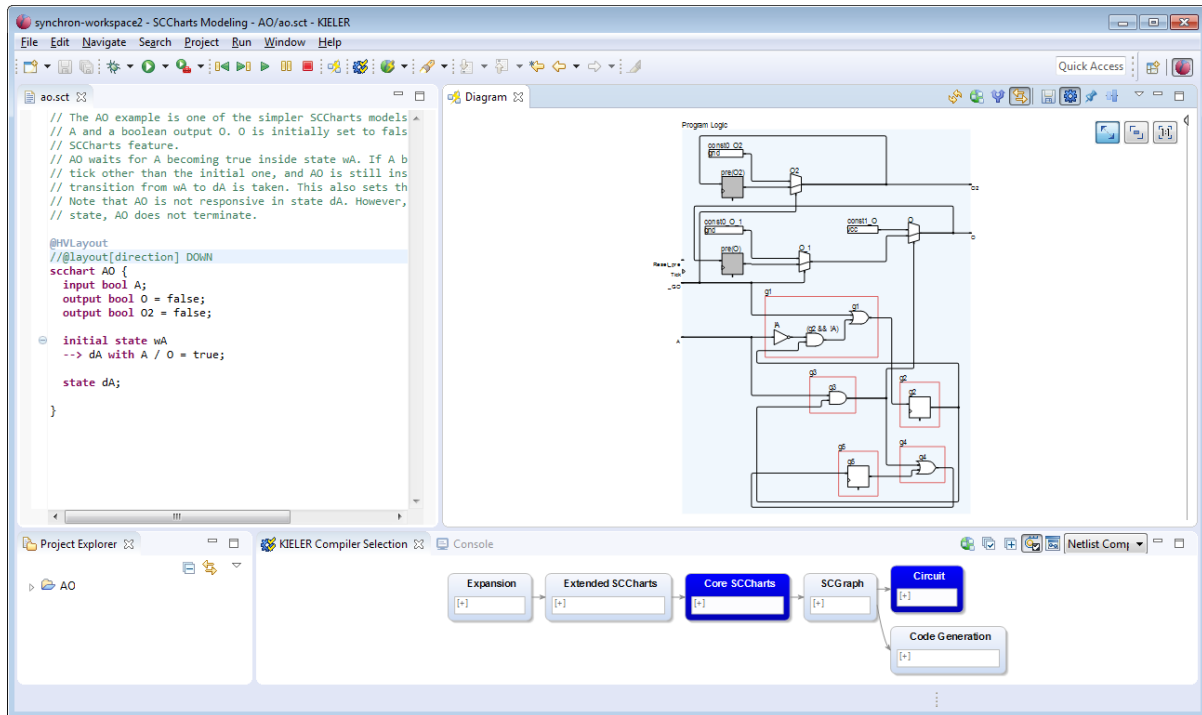
Note that the code generation will produce a `reset()` and a `tick()` function for every SCChart. The original interface is located in the top part of the variable declaration followed by internal guards for the circuit-based code generation.

Note that if you have opened the textual editor then the KIELER Compiler Selection view is disabled and will not react to mouse clicks

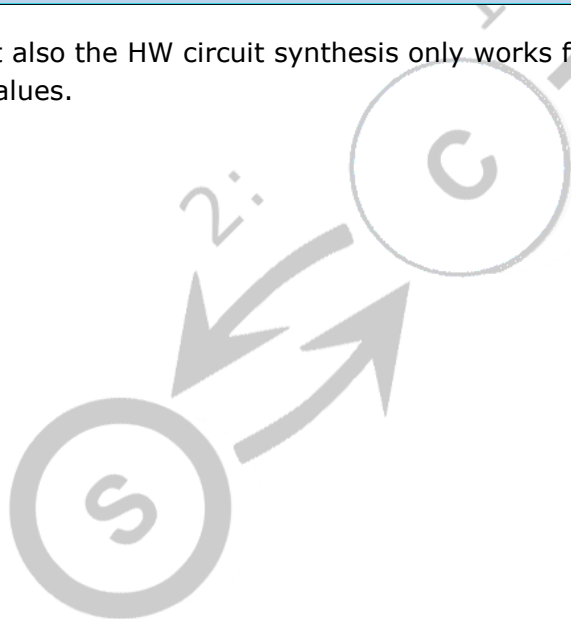


## 3.4 Generating Hardware Circuits

You can easily generate HW circuits by just clicking on the "Circuit" feature group.



Note that also the HW circuit synthesis only works for bool values and not for int or other kind of values.



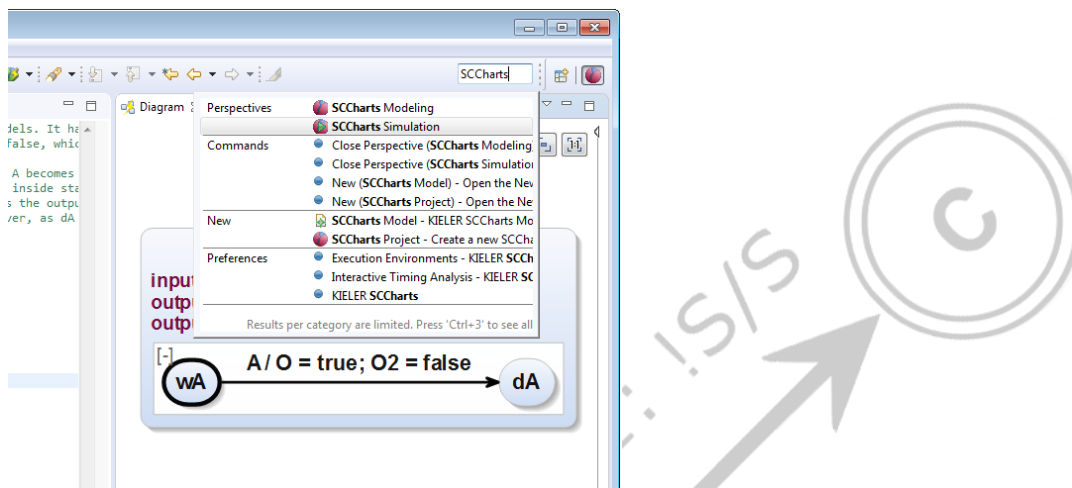
## 4. Exercise III: Simulation

Learning Objective: Familiarize yourself with the simulation feature of KIELER SCCharts.

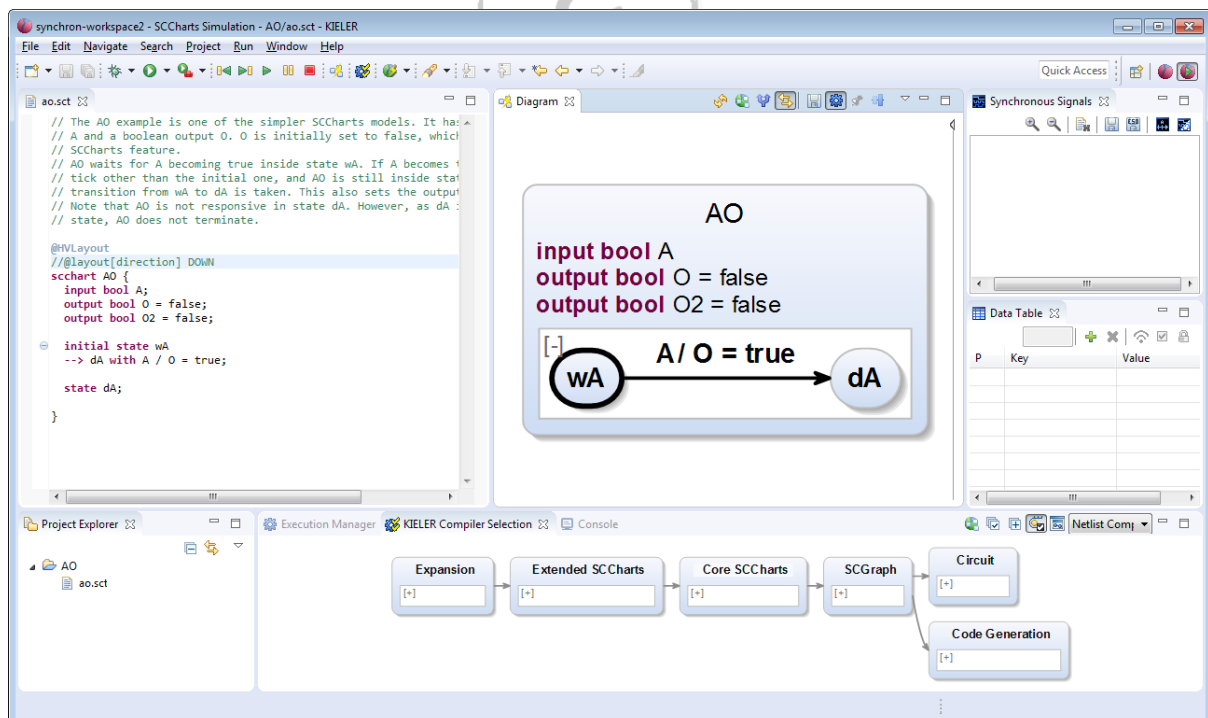
The simulation feature helps to understand the dynamics of an SCChart.

### 4.1 Simulation Perspective

There is a dedicated perspective for simulating SCCharts. You can reach it by starting to type "SCCharts" into the Quick Access textbox at the upper right corner of the window.

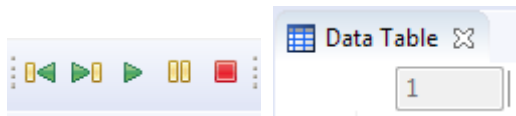


After choosing the simulation perspective the window should look like this:

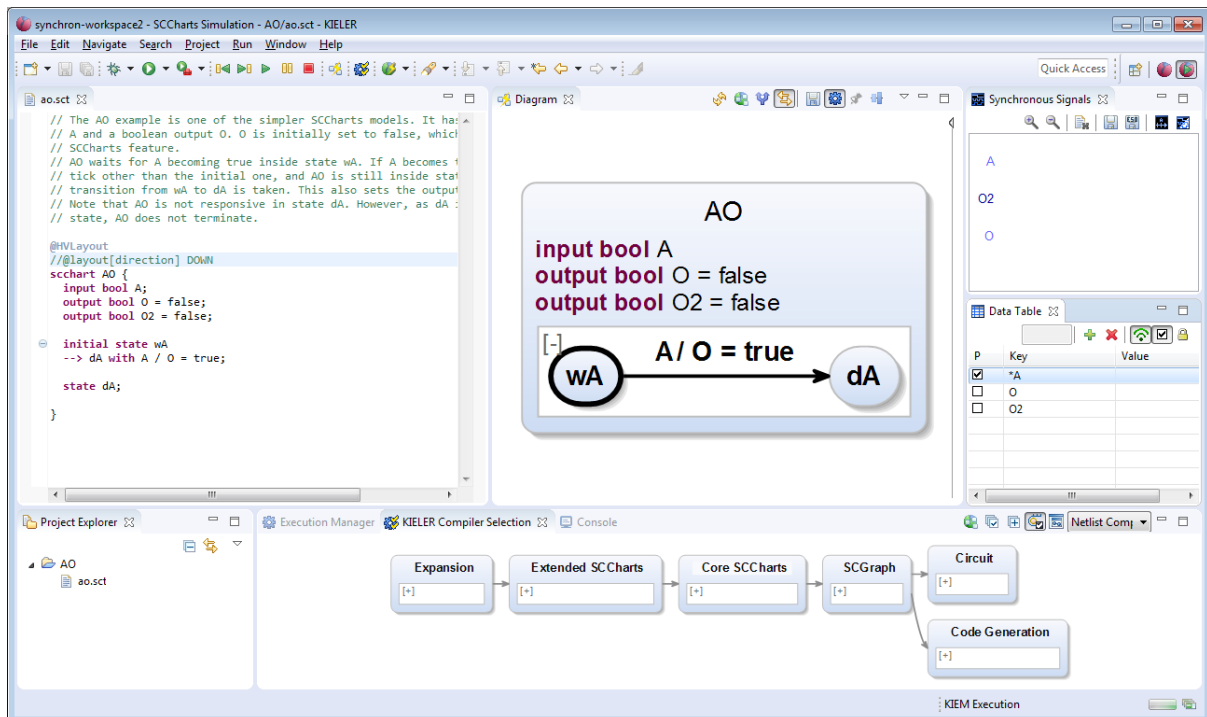


## 4.2 Simulation Control and Initialization

The synchronous step control buttons can be found in the toolbar. The Data Table has a text field which indicates the synchronous tick number.



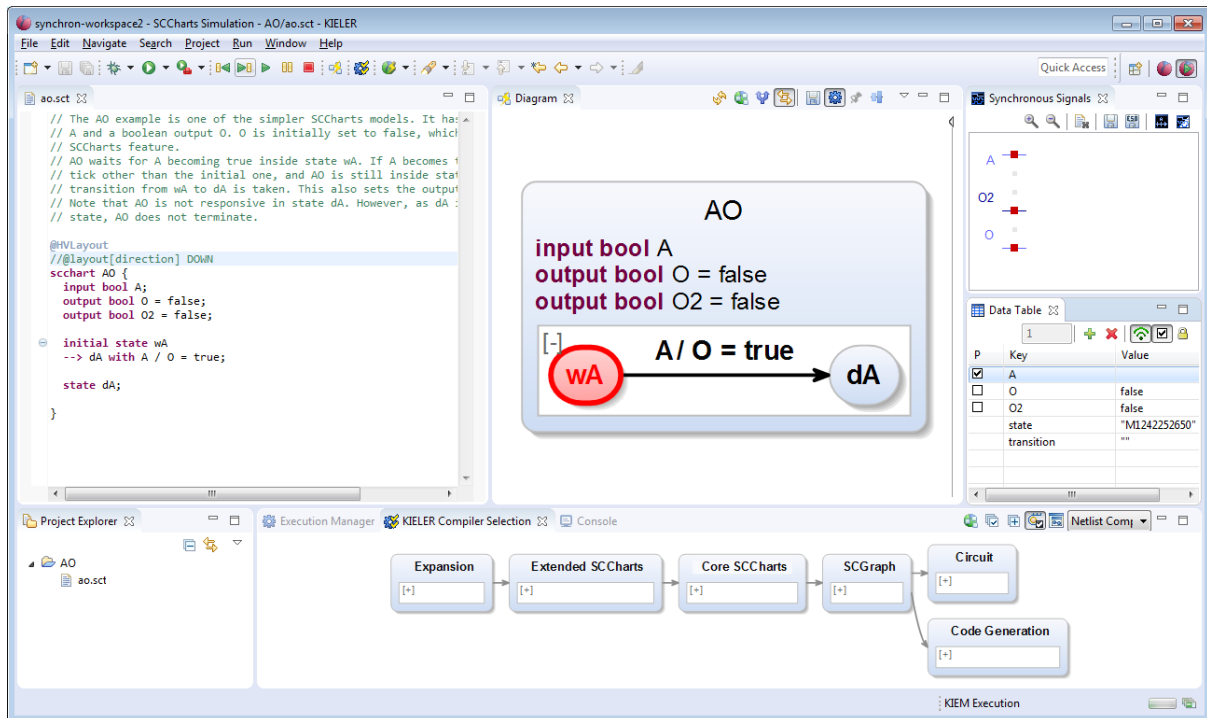
After clicking on the step button (the second button from left), the simulation is initialized before the first/initial tick.



The Data Table shows all interface variables and signals that are present in the model. The checkbox can be used for input signals to be set to present in the next tick or for input int/bool variables to set them to 1/true. The star (\*) marks all modified entries that are considered the for the next synchronous tick reaction calculation of the SCChart.

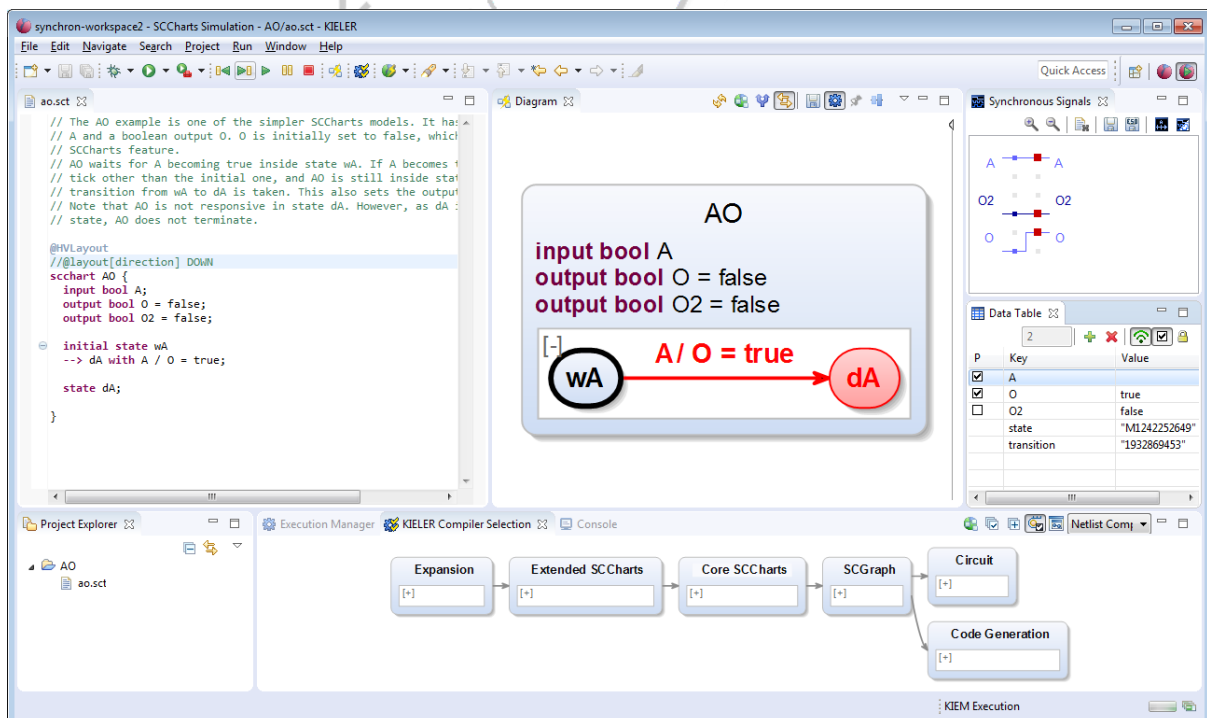
## 4.3 Synchronous Ticks

For example, one could set the input "A" to be true in the initial tick by checking the check box (see above screen shot). Then when performing the first real tick computation by clicking on the step button again, the active state wA is high-lighted as follows:



Note that AO did not react to the input "A" in the initial tick because it is modeled with a delayed transition.

If you set the input "A" for the next (second) tick and do another tick computation then the transition to "dA" is taken. Note that you have to click the checkbox of "A" twice in this scenario because the first click will modify "A" and unselect the checkbox and the second click will select the checkbox again. The simulation is also visualizing taken transitions as follows:



Note that the Synchronous Signals view above the Data Table shows the history of the signals and (bool/in) variable values according to the presence or true/false or 1/0 value.



## 4.4 Simulating Intermediate Models

Usually, the input for the simulation is the originally modeled SCChart as it was in the example above.

If one would like to investigate the dynamics of intermediate models during compilation then one needs again the KIELER Compiler Selection view. Stop any running simulation execution before selecting feature transformations.

E.g., select the "Core SCCharts" feature transformation. Note that this will include all necessary feature transformations from the "Extended SCCharts" feature group as well.

If you now initialize the simulation and for the initial/first tick initialize "A" with true (checked checkbox) then the behavior why AO does not react to "A" in the initial tick becomes more clear:

```

// The AO example is one of the simpler SCCharts models. It has
// A and a boolean output O. O is initially set to false, which
// is a SCCharts feature.
// AO waits for A becoming true inside state wA. If A becomes true
// on a tick other than the initial one, and AO is still inside state
// wA, the transition from wA to dA is taken. This also sets the output
// Note that AO is not responsive in state dA. However, as dA is the
// final state, AO does not terminate.

@HVLLayout
//@layout[direction] DOWN
scchart AO {
  input bool A;
  output bool O = false;
  output bool O2 = false;

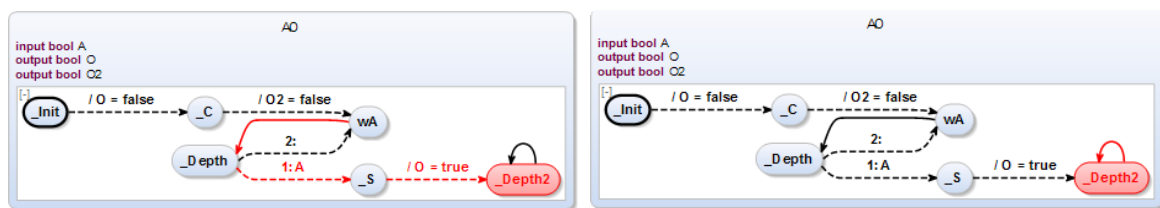
  initial state wA
  --> dA with A / O = true;

  state dA;
}
  
```

P	Key	Value
<input checked="" type="checkbox"/>	A	true
<input type="checkbox"/>	O	false
<input type="checkbox"/>	O2	false
	state	"M1242252647"
	transition	"1932869453,186"

All the initialization is done which sets the output "O" and "O2" both to false. But then the state "wA" is entered without checking for its outgoing transitions in the initial tick (because the only outgoing transition is a delayed and not an immediate transition).

Setting "A" again for the second tick results in the following simulation visualization diagram (left). The right diagram shows all following ticks in which AO will never terminate.



## 4.5 Hardware Circuit Simulation

For simulating HW circuits simply click on "Circuit" in the KIELER Compiler Selection view and then on the step button to initialize the simulation execution.

The screenshot shows the KIELER IDE interface for simulating an SCCharts model. The main window is titled "synchron-workspace2 - SCCharts Simulation - AO/ao.sct - KIELER".

**Code Editor (ao.sct):**

```

// The AO example is one of the simpler SCCharts models. It has
// A and a boolean output O. O is initially set to false, which
// SCCharts feature.
// AO waits for A becoming true inside state wA. If A becomes
// tick other than the initial one, and AO is still inside state
// transition from wA to dA is taken. This also sets the output
// Note that AO is not responsive in state dA. However, as dA
// state, AO does not terminate.

@HWLayout
//@layout[direction] DOWN
scchart AO {
  input bool A;
  output bool O = false;
  output bool O2 = false;

  initial state wA
  --> dA with A / O = true;

  state dA;
}
    
```

**Diagram View:** Shows a logic diagram with gates and multiplexers representing the hardware implementation of the SCCharts model.

**Synchronous Signals:** A timing diagram showing signals A, O2, O, and GO. Signal A is a square wave, O and O2 are initially low and become high when A transitions from low to high. GO is a single pulse.

**Data Table:**

P	Key	Value
<input type="checkbox"/>	_GO	0
<input checked="" type="checkbox"/>	A	1
<input type="checkbox"/>	g0	0
<input type="checkbox"/>	g1	0
<input checked="" type="checkbox"/>	g2	1
<input checked="" type="checkbox"/>	g3	1
<input checked="" type="checkbox"/>	out	1

**Compiler Selection View:** A flowchart showing the compilation process: Expansion → Extended SCCharts → Core SCCharts → SCGraph → Circuit. The "Circuit" step is highlighted in blue, indicating it is the selected target for simulation.

**Project Explorer:** Shows the project structure with "AO" and "ao.sct" files.

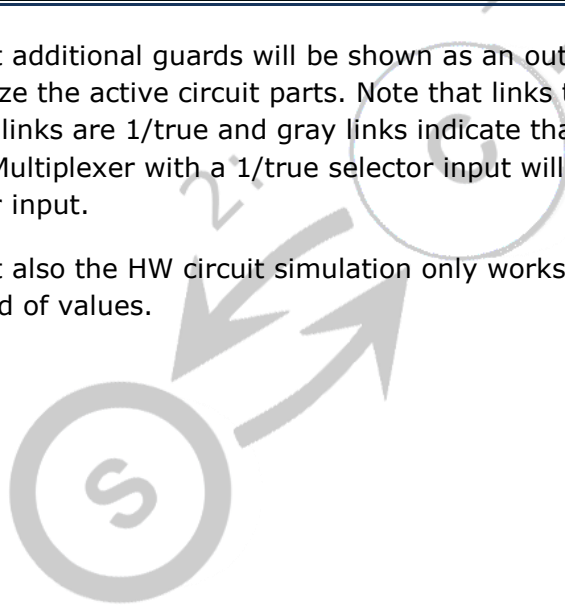
**Execution Manager:** Shows the "KIELER Compiler Selection" and "Console" tabs.

**Netlist Com:** A dropdown menu for selecting the netlist format.

**KIEM Execution:** A progress bar at the bottom right indicates the simulation execution status.

Note that additional guards will be shown as an output in the Data Table which are used to visualize the active circuit parts. Note that links that are red indicate that the values on these links are 1/true and gray links indicate that the values on these links are 0/false. Multiplexer with a 1/true selector input will forward their upper input, otherwise the lower input.

Note that also the HW circuit simulation only works for bool values and not for int or other kind of values.



## 5. Bonus Exercise IV: SCG

Learning Objective: Familiarize yourself with the control-flow graph representation of SCCharts, namely the Sequentially Constructive Graph (SCG).

Note: **If time permits**, you are invited to do this bonus exercise. The objective in detail is to understand the scheduling of the SCCharts compiler regarding initializations, updates, and reads (iur) of variables. In particular, this exercise shows how scheduling difficulties can be understood and resolved.

Recall that SCCharts distinguishes three kinds of variable accesses:

1. Absolute writes (also called "initializations"): These are always scheduled first.  
Example:  $X = 0$
2. Relative writes (also called "updates"): These are scheduled after all absolute writes. Example:  $X = X + 1$
3. Reads: These are always scheduled after all (absolute and relative) writes.  
Example:  $Y = X$

Note, there concurrent absolute writes cannot be scheduled and are not allowed. However, sequential absolute writes are scheduled as modeled. Concurrent relative writes can always be scheduled because of a commutative and associative combine function (e.g., ADD, OR, ...).

Consider the following SCChart:

The screenshot shows the SCCharts Modeling tool interface. The top-left pane displays the source code for chart `AO`:

```

scchart AO {
  input bool A;
  output int O = 0;

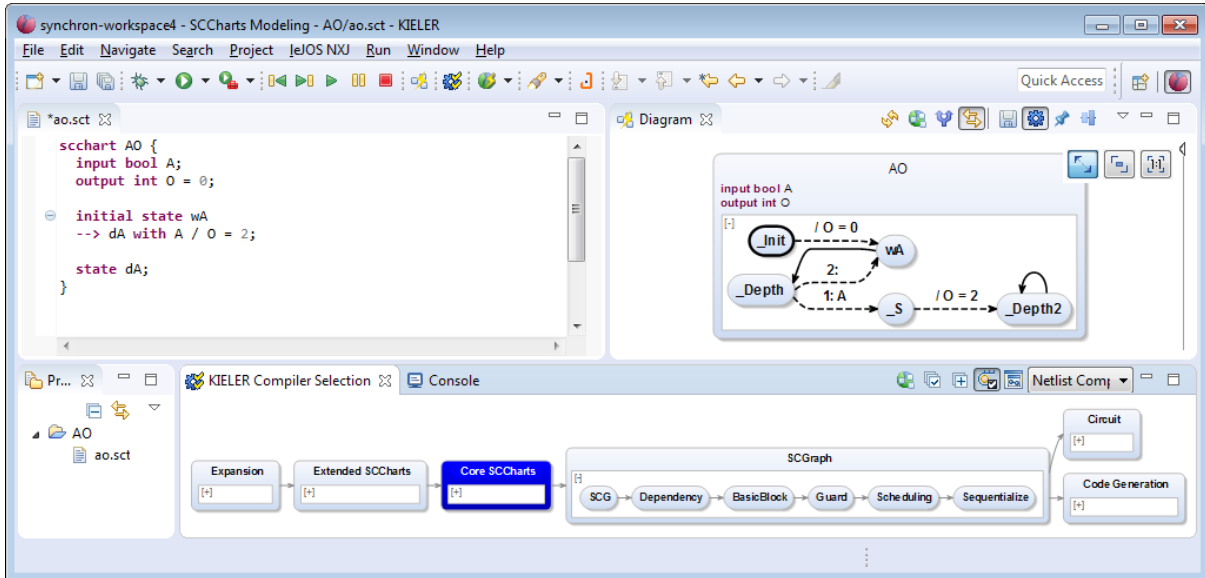
  initial state wA
  --> dA with A / O = 2;

  state dA;
}
    
```

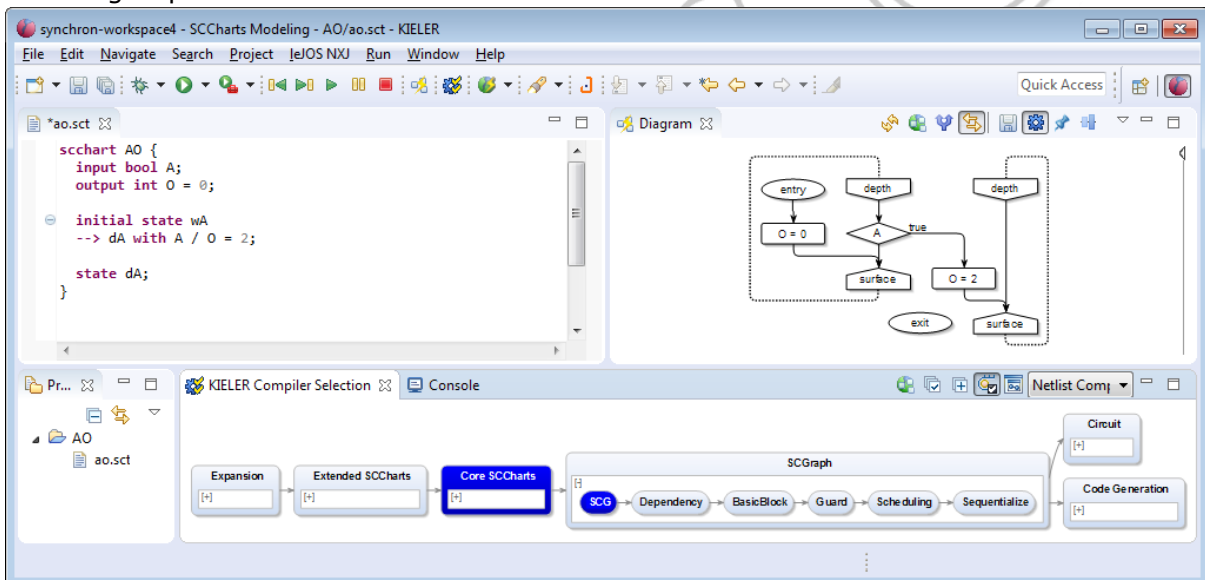
The top-right pane shows the diagram of the chart, which consists of an initial state `wA` and a transition `dA` triggered by `A / O = 2`. The bottom pane shows the compilation pipeline: `Expansion` → `Extended SCCharts` → `Core SCCharts` → `SCG` → `Dependency` → `BasicBlock` → `Guard` → `Scheduling` → `Sequentialize` → `Code Generation`.

Note that this can be scheduled without any problems although there are two absolute writes. The first sets  $O=0$  in the initialization and the second sets  $O=2$  in the transition. The modeled sequence becomes clear if one has a look at the normalized SCChart.

Select all "Core SCCharts" transformations in the "Compiler Selection" view:

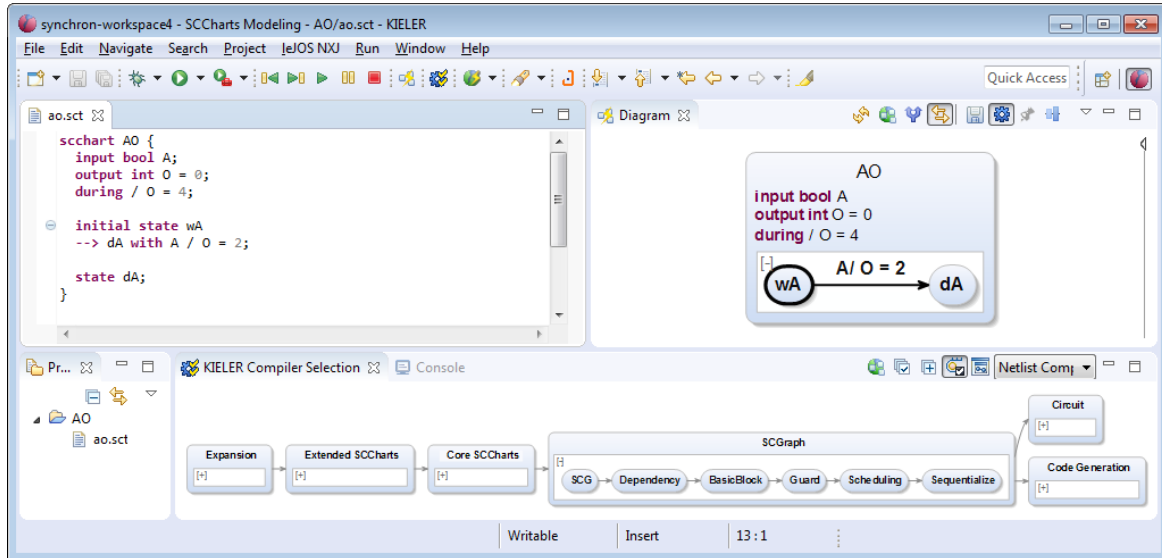


This can be directly mapped to the SCG representation. Select "SCG" in the "SCGraph" feature group:

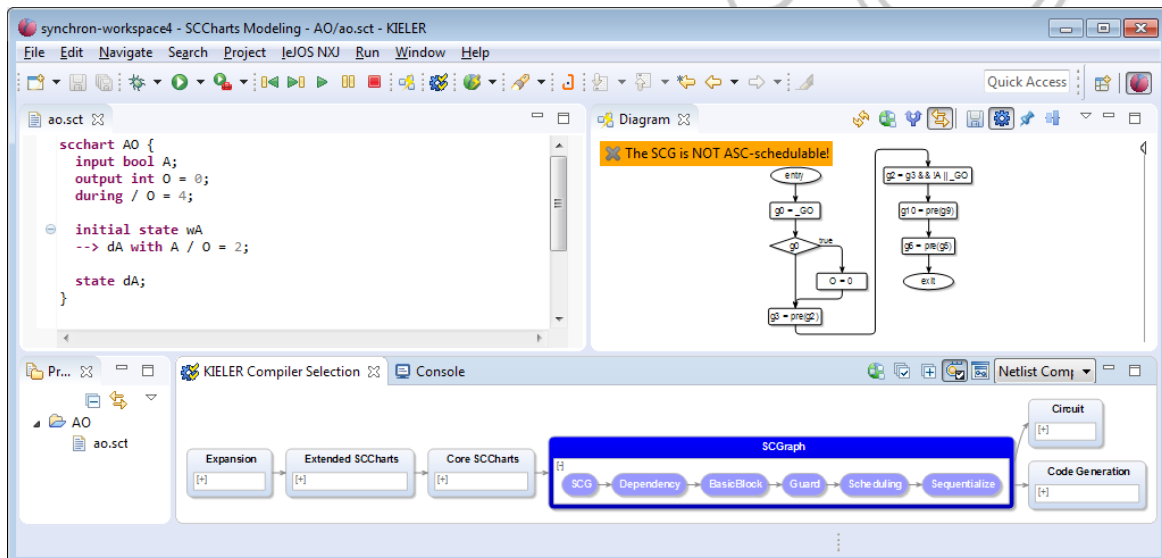


The "entry" node is the start of the (single) thread here. If started, first O is initialized to 0 then a pause-construct ("surface" + "depth") is entered. In the next tick this pause-construct is left. Then, "A" is checked. If "A" is false (else branch), then the pause-construct is entered again. If "A" is true (then branch), then the O is set to 2 and another pause-construct (of the other former state "dA") is entered. As the program never terminates, the "exit" node is left unconnected.

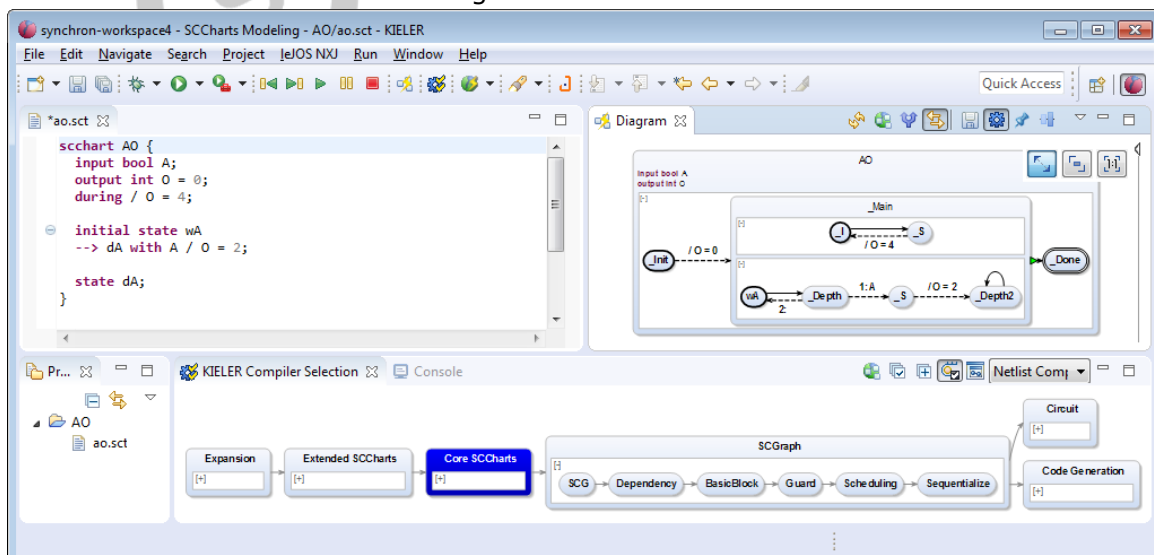
Now modify the original SCChart and add a during action that sets O to 4:



If you now select all "SCGraph" transformations, then you notice that the SCG is not ASC-schedulable:

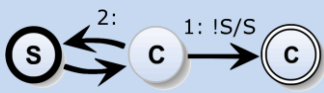


To investigate the problem, take a look at the normalized SCChart, selecting only all "Core SCCharts" transformations again:



The initialization of  $O=0$  is still not a problem. But the during actions is executed concurrently such that this results in a concurrent  $O=4$  and  $O=2$  absolute write. Looking at the SCG you can visually see the dependencies and this so called **write-write-conflict**:

A solution is to make one of these absolute writes a relative write, e.g., using addition as a proper combination function:



As expected, the absolute write is now scheduled before the concurrent relative write. And the SCG is now properly schedulable:

```

scchart AO {
  input bool A;
  output int O = 0;
  during / O = O + 4;

  initial state wA
  --> dA with A / O = 2;

  state dA;
}
  
```

Diagram components:

- entry node
- Guard  $g0 = \_GO$
- Decision  $g0$  (true) leading to assignment  $O = 0$
- Guard  $g2 = \_GO \parallel g3 \ \&\& \ !A$
- Decision  $g4 = g3 \ \&\& \ A$  (true) leading to assignment  $O = 2$
- Guard  $g10 = \text{pre}(g9)$
- Decision  $g10b = g10$  (true) leading to assignment  $O = O + 4$
- Guard  $g9 = \_GO \parallel g10b$
- exit node
- Preconditions:  $g3 = \text{pre}(g2)$ ,  $g6 = \text{pre}(g5)$ ,  $g5 = g6 \parallel g4$

Dependency Filter:

- write - write
- Sausage Folding
- abs. write - rel. write
- write - read
- rel. write - read

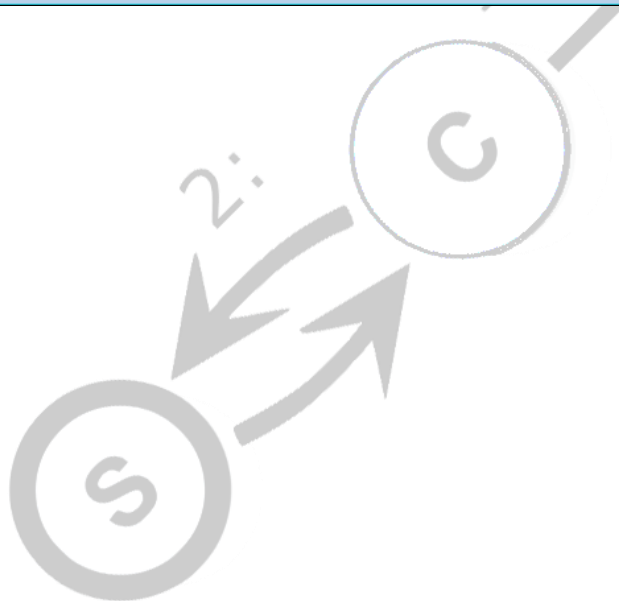
Alignment:

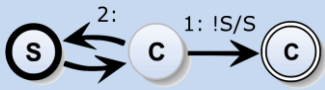
- Tick start

Compiler Selection Flowchart:

```

Expansion -> Extended SCCharts -> Core SCCharts -> SOGraph
SOGraph -> SCG -> Dependency -> BasicBlock -> Guard -> Scheduling -> Sequentialize
SOGraph -> Circuit
SOGraph -> Code Generation
  
```





## 6. Lego Mindstorms Preparations

Before we are able to use SCCharts with Lego Mindstorms, some preparation is necessary.

**Please note that most limitations that are described here are limitations of leJOS or its Eclipse plugin, not of the KIELER SCCharts tools.**

If you do not want to struggle with these extra preparations, then you are invited to use our provided **DOWNLOAD STATION**. Please use one of the USB sticks to transfer your SCT textual SCChart file to the Download Station.

Note that the installation procedure varies depending on your operating system. In essence, two steps are necessary.

1. Lego Mindstorms driver
2. LeJOS project

First, the Lego device needs to be registered in your operating system. Then the leJOS project needs to be installed. This provides a Java capable firmware for the Lego brick and the uploading facility. The leJOS plugin for Eclipse should be available in the KIELER SCCharts tools already. It will use the installed leJOS project to compile, upload, and run the Java files that are generated from SCCharts.

### 6.1 Windows

For Windows you will need the **32bit** version of both (a) KIELER SCCharts and (b) Java (see Section 1). Make sure that the 32 Bit Java version is available for the KIELER SCCharts RCA. You may need to check your PATH variable or edit the kieler.ini of the SCCharts RCA in rare cases.

#### USB Driver (Fantom)

For Windows you will need the Lego Fantom Driver which you may download here or install from the provided USB stick:

<https://mi-od-live-s.legocdn.com/r/www/r/mindstorms/-/media/franchises/mindstorms%202014/downloads/firmware%20and%20software/nxt%20software/nxt%20fantom%20drivers%20v120.zip>

or

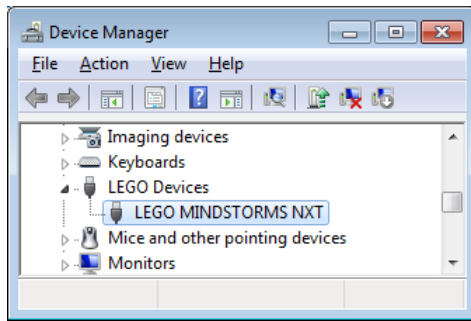
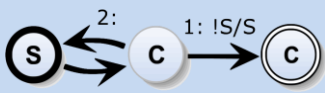
<http://tinyurl.com/legodriver>

or

the file /USBStick/WindowsMacFantomDriver/**NXT Fantom Drivers v120.zip**  
(from the USB Stick)

After the installation, connect the Mindstorms brick with a USB cable. Check whether you see the Lego Mindstorms device in the Windows Device Manager:





## Install LeJOS

You will further need leJOS which you will get from the USB stick or from the following URL: <https://sourceforge.net/projects/nxt.lejos.p/files/>

or use the following tiny URL: <http://tinyurl.com/lejosinstall>

Note, that you will need a version that fits to the flashed firmware on the brick. Currently most of our bricks run with **version 0.9.1**.

For Windows, you will mostly like to run the `/USBstick/leJOS/leJOS_NXJ_0.9.1beta-3_win32_setup.exe` executable.

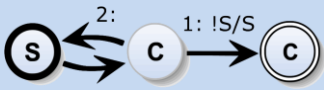
## 6.2 Mac OSX

For MacOSX you need to start KIELER SCCharts from the command line. You need to set two path variables before (for details, see below), namely `NXJ_HOME` and `LEJOS_NXT_JAVA_HOME`.

Further, leJOS needs a 32 Bit Java and for MacOSX the most recent 32 Bit Java is Version 1.6. Hence, you need to install Java 1.6 in addition from Oracle Homepage or from the USB stick.

For KIELER SCCharts you still need Java 1.8.

If you need to install Java 1.6 and Java 1.8, please install Java 1.8 first and then Java 1.6 as the Java 1.8 installation procedure is known to possibly corrupt a previously installed Java 1.6 on MacOSX.



## USB Driver (Fantom)

For MacOSX you will need the Lego Fantom Driver which you may download here or install from the provided USB stick:

<https://mi-od-live-s.legocdn.com/r/www/r/mindstorms/-/media/franchises/mindstorms%202014/downloads/firmware%20and%20software/nxt%20software/nxt%20fantom%20drivers%20v120.zip>

or

<http://tinyurl.com/legodriver>

or

the file /USBStick/WindowsMacFantomDriver/**NXT Fantom Drivers v120.zip**  
(from the USB Stick)

## Install LeJOS

The OS X edition of leJOS is distributed as a .tar.gz archive. Untar the archive to a location of your choice. You may delete the build subfolder, since it is required for Linux only.

```
tar -xvzf leJOS_NXJ_0.9.1beta-3.tar.gz
```

Set the NXJ\_HOME path variable in the KIELER/Eclipse preferences and let it point to your leJOS installation (the root folder, not the bin folder). See explanation in the end.

Also set the leJOS path variable for your shell:

```
export NXJ_HOME="<your leJOS installation root directory>"
```

## Install Java Version 1.6 32 Bit

For MacOSX, you need a 32 Bit version of Java. The latest version of Java that supports 32 Bit on a Mac is 1.6. Unfortunately, all newer version do not even support a 32 Bit mode (which often could be enabled using the -d32 parameter).

Hence, you need to download Java 1.6 32 Bit and install it parallel to you Java 1.8, which you need for working with KIELER SCCharts.

You get the installation bundle here:

[http://support.apple.com/downloads/DL1572/en\\_US/javaforosx.dmg](http://support.apple.com/downloads/DL1572/en_US/javaforosx.dmg)

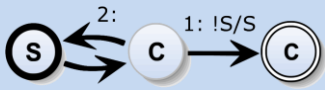
or

<http://tinyurl.com/java16mac>

or from the provided USB stick.

For leJOS to find Java 1.6, you need to set an environment variable.

```
export LEJOS_NXT_JAVA_HOME = "<your Java 1.6 installation root directory>"
```



## Start KIELER w/ 32Bit

You need to set the two path variables `NXJ_HOME` and `LEJOS_NXT_JAVA_HOME` before you start KIELER SCCharts from the command line.

Then, navigate to the folder where you expanded KIELER SCCharts and start it using

```
./kieler
```

## 6.3 Linux

### Prepare Java

If you don't have a JDK yet, download and install the newest release from oracle for your system or use the Java SDK provided on the USB stick. LeJOS needs to know the location of this JDK. This can be achieved by creating an environment variable `JAVA_HOME`. On most Linux systems, environment variables are edited in a file `.profile` in your home directory. Thus add

```
export JAVA_HOME="/opt/java/jdk1.8.0_60"
```

to the end of `~/profile`. Thereby the path to java must be adapted to your system. Logout and login again to let the change take effect. You can test this by typing `echo $JAVA_HOME`. The command should print the path you configured.

### USB Driver

Linux systems require the packages **libusb** and **libusb-dev** to connect via USB with a Mindstorms brick. On Ubuntu, these can be installed from command prompt using

```
sudo apt-get install libusb-0.1 libusb-dev
```

To use the USB connection as non-root user, create a file `/etc/udev/rules.d/70-lego.rules` with the following content (4 lines in total):

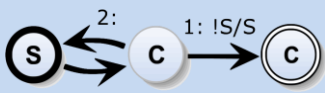
```
# Lego NXT brick in normal mode
SUBSYSTEM=="usb", DRIVER=="usb", ATTRS{idVendor}=="0694", ATTRS{idProduct}=="0002", GROUP="lego", MODE="0660"
# Lego NXT brick in firmware update mode (Atmel SAM-BA mode)
SUBSYSTEM=="usb", DRIVER=="usb", ATTRS{idVendor}=="03eb", ATTRS{idProduct}=="6124", GROUP="lego", MODE="0660"
```

Then create a group named **lego** and add your user to this group using the following commands:

```
sudo groupadd lego
```

```
sudo gpasswd -a <username> lego
```

```
sudo udevadm control --reload-rules
```



## Install LeJOS

You will further need leJOS, which can be downloaded from the following URL:  
<https://sourceforge.net/projects/nxt.lejos.p/files/>

Note, that you will need a version that fits to the flashed firmware on the brick. Currently, most of our bricks run with **version 0.9.1**.

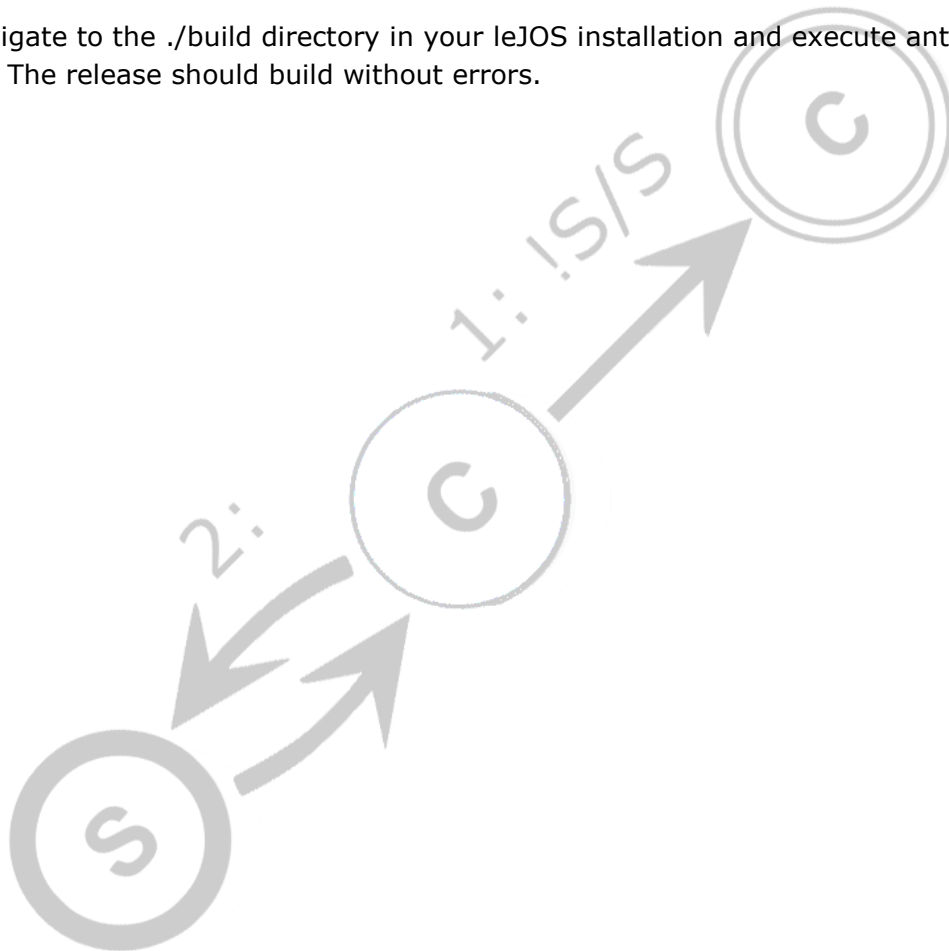
For Linux, you have to download and extract the tar.gz file:

```
tar -xvzf leJOS_NXJ_0.9.1beta-3.tar.gz
```

Afterwards you will need to build the Java Native Library that ships with leJOS. To do so, you will need ant.

```
sudo apt-get install ant
```

Then navigate to the ./build directory in your leJOS installation and execute ant from terminal. The release should build without errors.



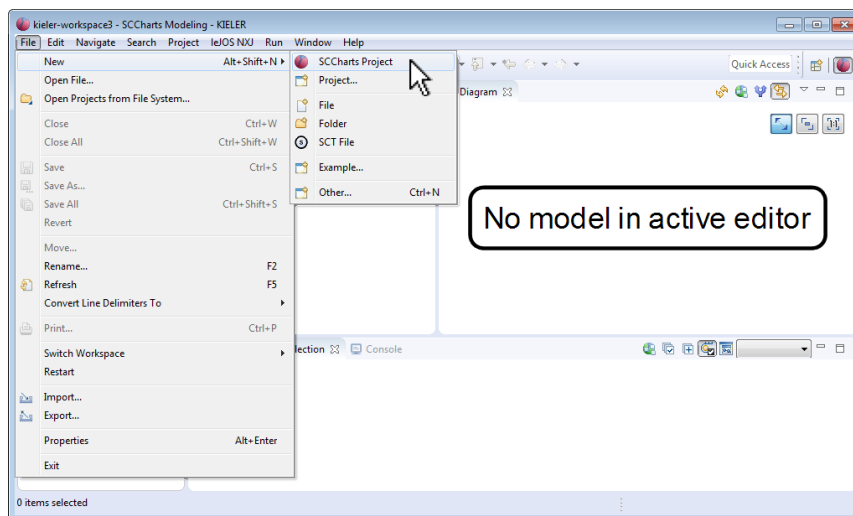
## 7. Exercise V: SCCharts for Lego Mindstorms

Learning Objective: Familiarize yourself with the SCCharts template environment using the example of Lego Mindstorms as an example embedded target.

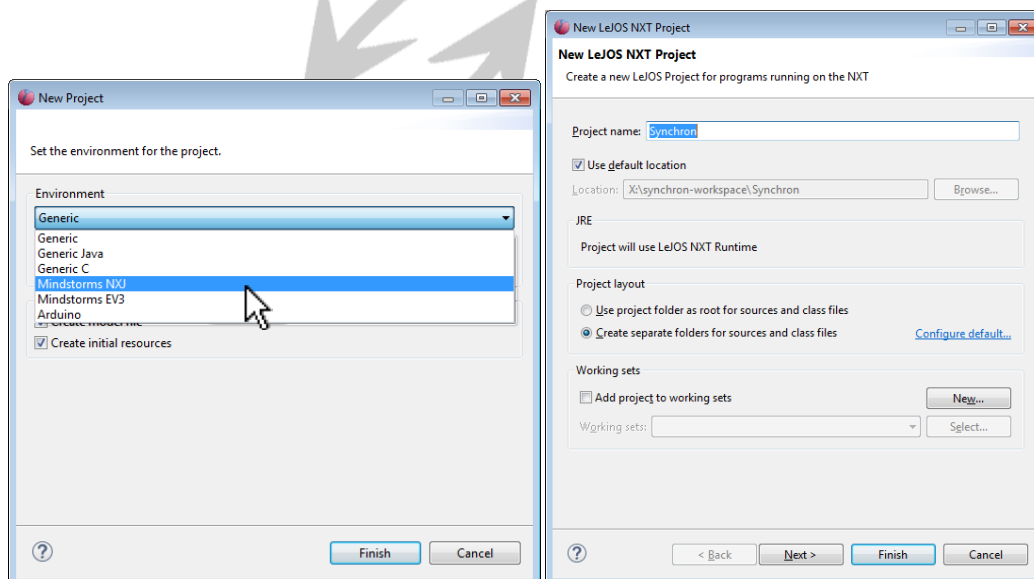
In this exercise you will create a simple SCChart which will wait on the OK button of the Lego Midstorms and output "Hello Synchron" to its display.

### 7.1 Creating an SCCharts Project

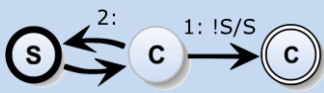
Click on "File" -> "New" -> "SCCharts Project" in the main menu.



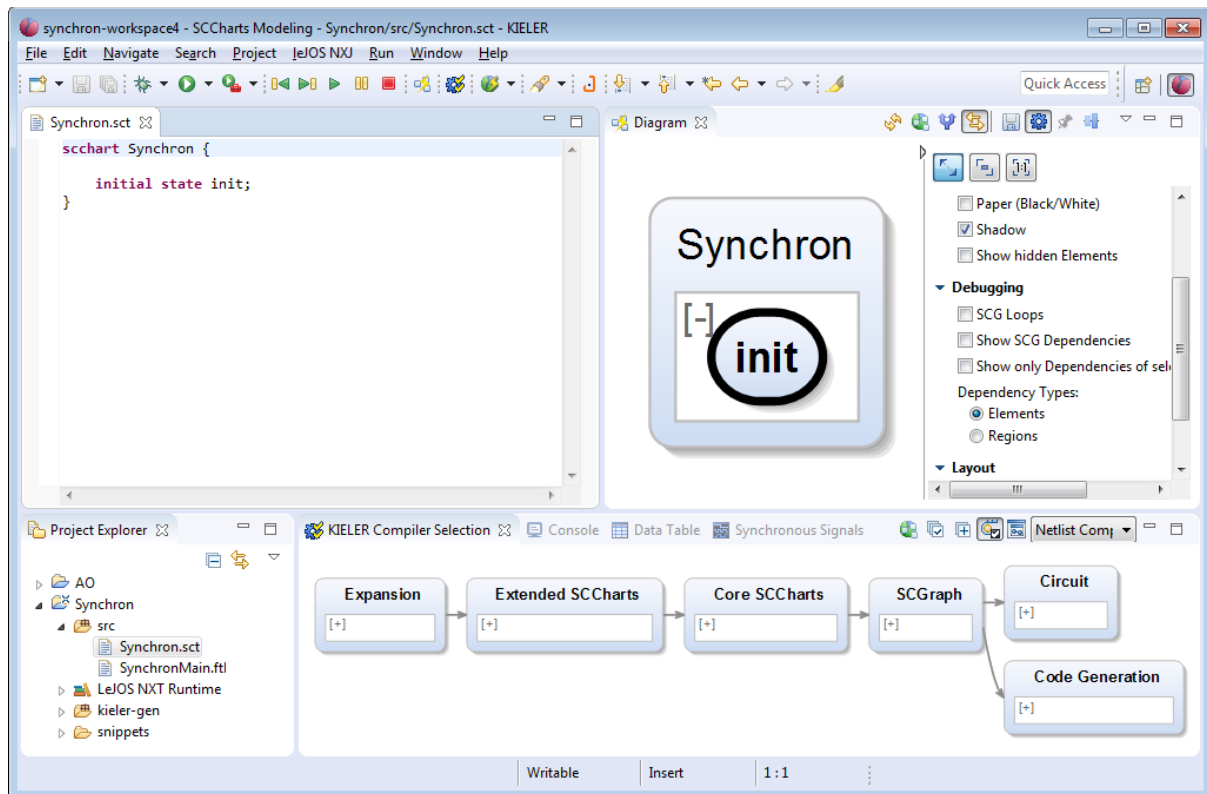
The select "Mindstorms NXJ" and click on "Finish" (left figure). Note that on MacOSX you need to select "Mindstorms NXJ (MacOSX)".



You will be prompted another New Java Wizard now (right figure). Enter a project name, e.g., "Synchron" and again click on "Finish". **Do not switch to the Java perspective, if you are prompted to do so. Attention: Due to leJOS limitations the name of the project must not exceed 8 characters!**

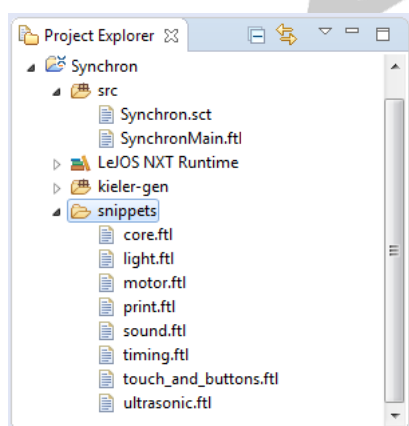


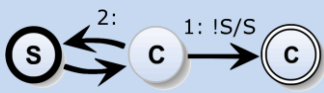
The screen should look as follows:



## 7.2 Modeling SCCharts with Environment Snippets

You can now start modeling with SCCharts and its Lego Environment Extension. Note the "snippets" folder where you can draw from various Lego I/O examples.





Modify the example as follows:

```

scchartSynchron {
    @Wrapper Button, ENTER
    input bool isEnterDown;

    @Wrapper Print
    output string outputText;

    initial state init
    --> done with isEnterDown / outputText = "Hello Synchron";

    state done;
}
  
```

The wrapper annotation "Enter" will enforce that the automatically generated environment wrapper code will be mapped to the "isEnterDown" boolean variable such that "isEnterDown" is true iff the Lego Mindstorms Enter button is pressed. The wrapper annotation "Print" will enforce that the automatically generated environment wrapper code will print the content of the "outputText" string variable on the Lego Minstorms display unit. Further snippets are available in the "snippets" folder (see above). Each snippet is a Freemarker template and comes with a short example.

### Further Snippet Examples

Light-Sensor:

```

@Wrapper LightSensor, S3
input int light;
  
```

Motor-Actuator:

```

@Wrapper MotorSpeed, A
@Wrapper MotorSpeed, B
output int speed;
  
```

Time Input:

```

@Wrapper Clock, "1000"
input bool second;
  
```

Touch Sensor:

```

@Wrapper TouchSensor, S1
input bool GO;
  
```

Lamp:

```

@Wrapper RCXLamp, A
output bool blink;
  
```

Beep:

```

@Wrapper Beep
output bool beep;
  
```

Draw String to pos (x,y):

```

@Wrapper DrawString, "0", "2"
output string blackString = "black: ";
  
```

Draw Int to pos (x,y):

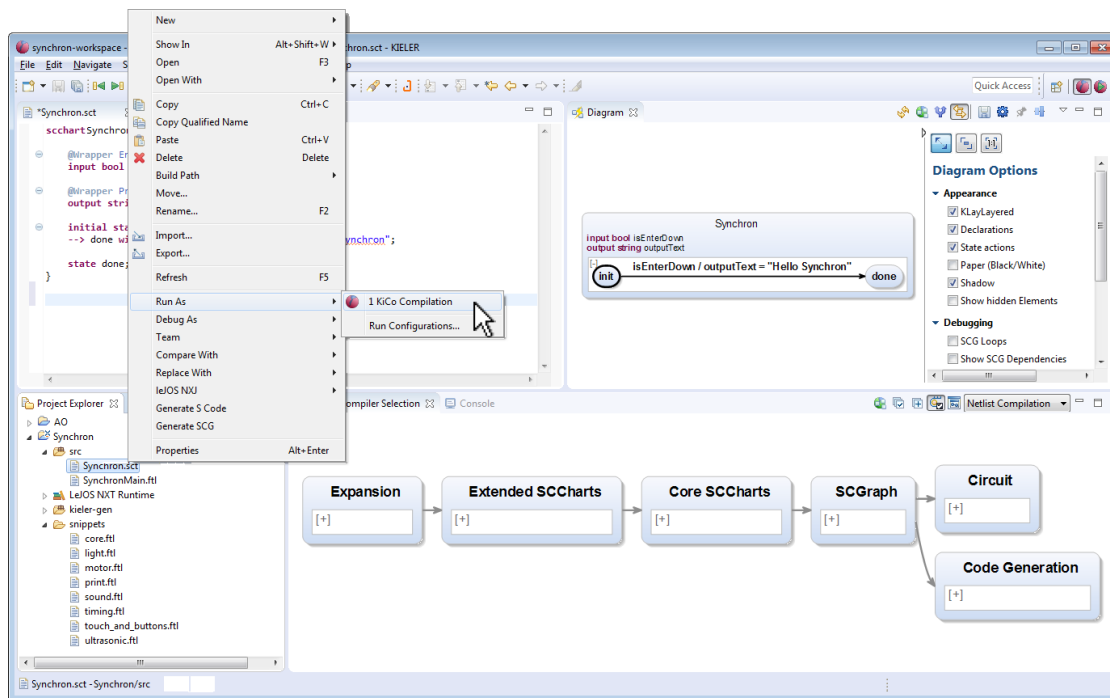
```

@Wrapper DrawInt, "8", "2"
output int blackValue;
  
```

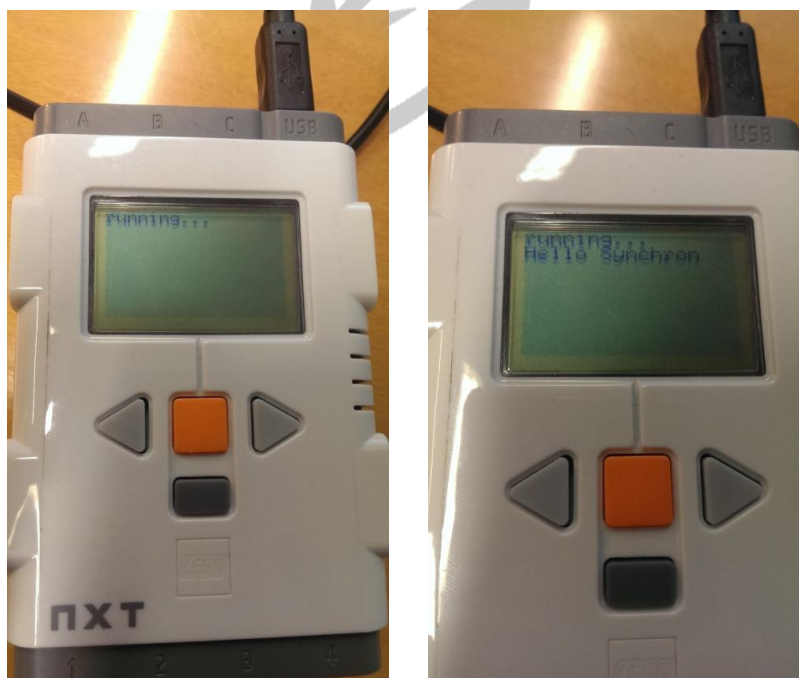
## 7.3 Download & Run

Please ensure that the **Lego Mindstorms brick is switched on now** and connected via a USB connection.

To upload right-click the sct file and select "Run As"->"KiCo Compilation" (see next page). This will automatically generate a Java file for the SCChart under a new "kieler-gen" folder. It will further generate environment wrapper code and try to upload and run the class files to the Lego brick.



The program should be uploaded to the brick and started. You should see the following screen on the display unit (left photo):



After pressing the (orange) Enter button, the output "Hello Synchron" should appear on the display unit (right photo).



## If you encounter Problems

### 1. If you get the following error:

Linking ...

Program has been linked successfully

Uploading ...

```
leJOS NXJ> Searching for any NXT using Bluetooth inquiry
```

```
BlueCove version 2.1.0 on winsock
```

```
leJOS NXJ> Failed to find any NXTs
```

```
leJOS NXJ> Failed to connect to any NXT
```

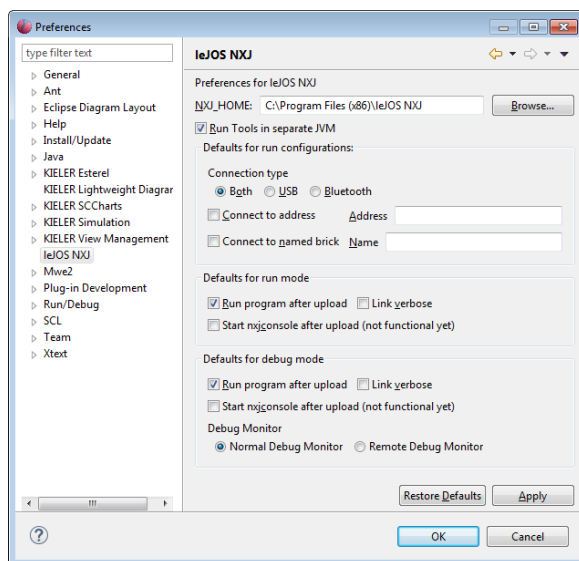
```
No NXT found - is it switched on and plugged in (for USB)?
```

Then most likely the NXT is not turned on (any more) or it is not connected via USB.

Note that the NXT switches off automatically after some time.

### 2. If you have problems uploading your SCCharts to the Lego Mindstorms brick, you may need to check the LeJos configuration in the preference settings.

Click on "Window" -> "Preferences" and select "leJOS NXJ". Then, check the path to the leJOS installation.



## 8. Exercise VI: SCCharts Pathfinder

Learning Objective: Use your knowledge about SCCharts and the KIELER SCCharts tooling to solve a more complex embedded system task.

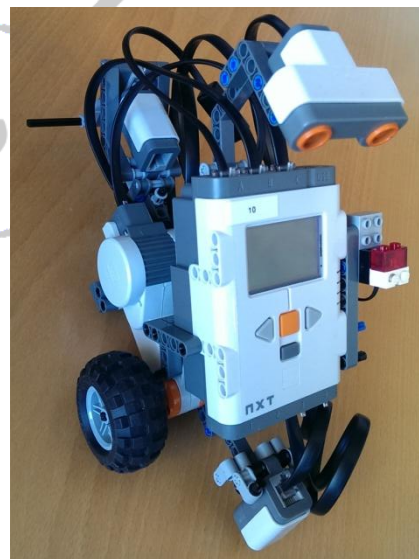
**This exercise is not meant to be completely solved in the short time frame of this tutorial workshop slot.**

However, you are invited to solve it during the SYNCHRON workshop until Friday morning whenever time permits (breaks, evenings, ...). We will leave the robots, the Download Station, and the mat for your pleasure (hopefully ☺). We may assist you any time; just let us know.

### 8.1 The Task

We brought several Lego Mindstorms that were build by some of your students. Additionally, we brought you an exercise mat with a printed path.

Your task is to build an SCCharts controller for the Mindstorms robot such that the robot follows the line from "Start" to "Finish" reliably and as fast as possible.



### 8.2 Your Solution

We would be happy if you share your solution with us. You are happy to add comments about what you liked and what you would like us to improve in the SCCharts language and tooling.

We would appreciate if you send us your solution via email to:

[kieler@informatik.uni-kiel.de](mailto:kieler@informatik.uni-kiel.de)

## 9. Questions & Feedback

We'd like to encourage you to give us your valuable feedback on anything that comes to your mind when using the KIELER SCCharts tools.

### 9.1 Questions

For questions, you may use the following link that takes you to our Confluence Questions section:

<https://rtsys.informatik.uni-kiel.de/confluence/questions>

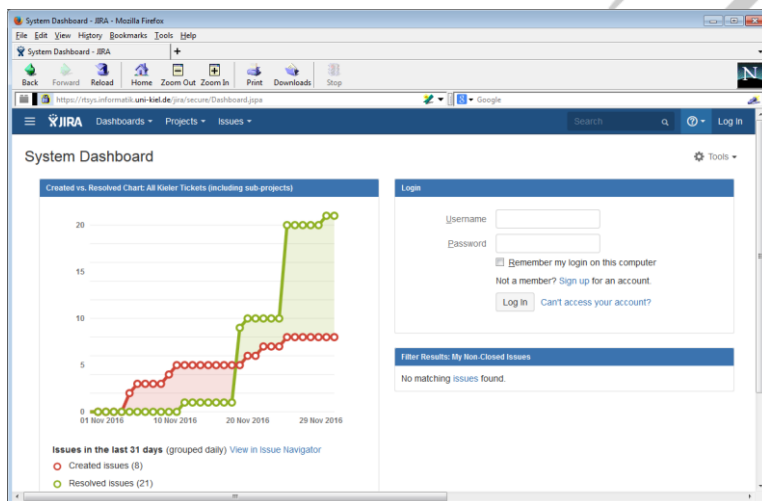
or

<http://tinyurl.com/kielerquestions>

### 9.2 Report Errors

If you encounter any errors or faulty behavior or if you have concrete feature requests, then you may use our Jira bug tracker here:

<https://rtsys.informatik.uni-kiel.de/jira/> or <http://tinyurl.com/kielerbug>



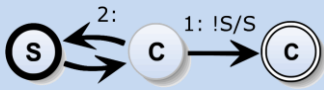
If you do not have an account yet, please sign up for one. If you do not like to open an account, you may also send us your bug report via email:

**[kieler@informatik.uni-kiel.de](mailto:kieler@informatik.uni-kiel.de)**

### 9.3 Feedback

We'd also like to hear about your valuable feedback in general. We kindly invite you to participate in the following survey (see next page).

Optionally, you may also send us an email: [kieler@informatik.uni-kiel.de](mailto:kieler@informatik.uni-kiel.de)



# SCCHARTS SURVEY FORM

Have you use KIELER SCCharts **before**?

- Yes**  
Please let us know in the comments below, where you used it.
- No**

Can you image to use KIELER SCCharts in the future?

- Yes**     in class     in projects     for fun
- No**

Rate the quality of the SCCharts development tools you worked with.

Creation/**Modeling** of models:

- Professional**  
as other mature open source or commercial products
- Advanced**  
as other smaller commercial or open source products
- Ok**  
as beta version of commercial software or Freeware
- Hardly usable**  
like alpha versions or private/obsolete projects

**Debugging** of models:

- Professional**  
as other mature open source or commercial products
- Advanced**  
as other smaller commercial or open source products
- Ok**  
as beta version of commercial software or Freeware
- Hardly usable**  
like alpha versions or private/obsolete projects

**Interactive** compilation/code generation:

- Professional**  
as other mature open source or commercial products
- Advanced**  
as other smaller commercial or open source products
- Ok**  
as beta version of commercial software or Freeware
- Hardly usable**  
like alpha versions or private/obsolete projects

**Wrapper\Template** code generation:

- Professional**  
as other mature open source or commercial products
- Advanced**  
as other smaller commercial or open source products
- Ok**  
as beta version of commercial software or Freeware
- Hardly usable**  
like alpha versions or private/obsolete projects

**Understanding** the language semantics:

- Professional**  
as other mature open source or commercial products
- Advanced**  
as other smaller commercial or open source products
- Ok**  
as beta version of commercial software or Freeware
- Hardly usable**  
like alpha versions or private/obsolete projects

**User interface:**

- Professional**  
as other mature open source or commercial products
- Advanced**  
as other smaller commercial or open source products
- Ok**  
as beta version of commercial software or Freeware
- Hardly usable**  
like alpha versions or private/obsolete projects

**Comments** on KIELER SCCharts:

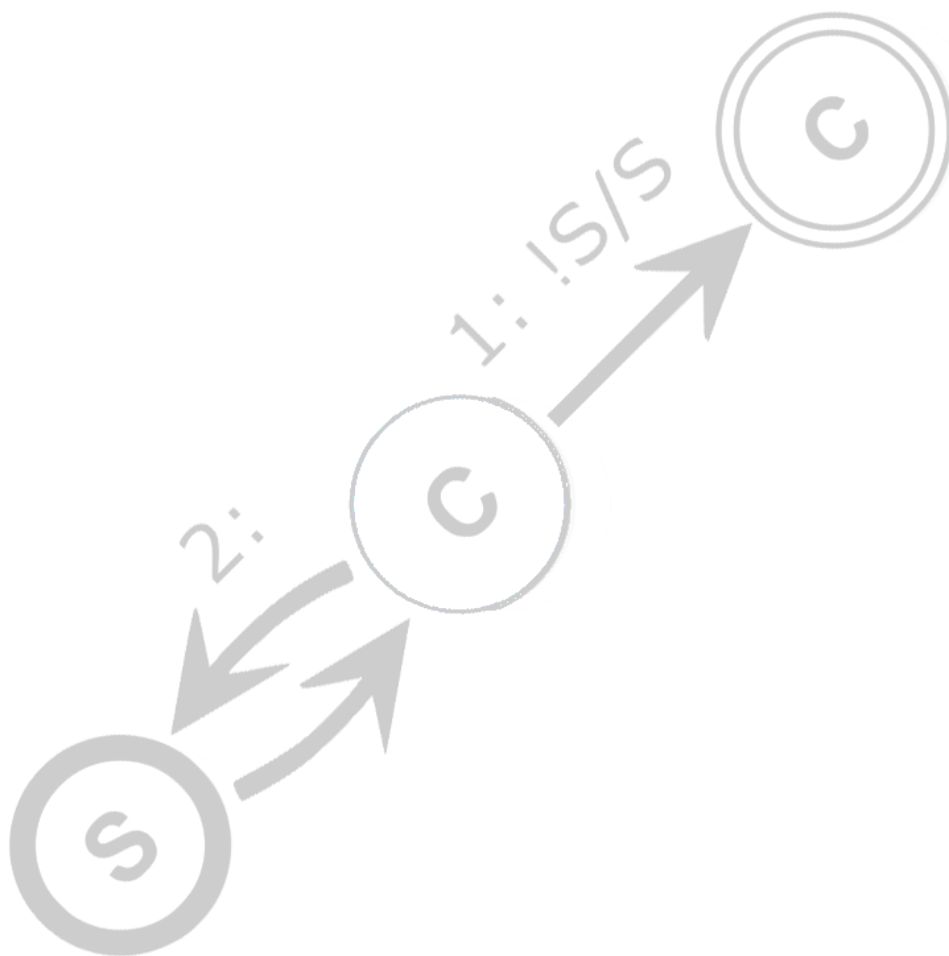
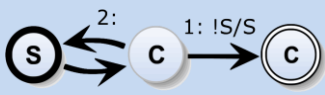
(E.g., what do you think is missing for the release of the KIELER SCCharts development tools?)

**Comments** on this tutorial:

(E.g., what can we do to improve it? Was it helpful? Did you enjoy it?)

You can use the back page of this form to leave more detailed comments and/or suggestions.

**We appreciate your feedback! Thank you very much!**

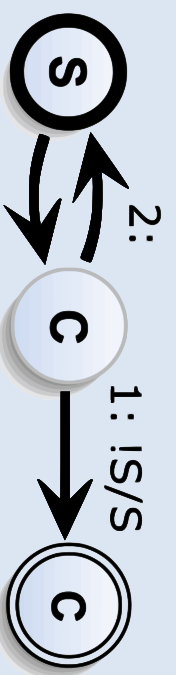
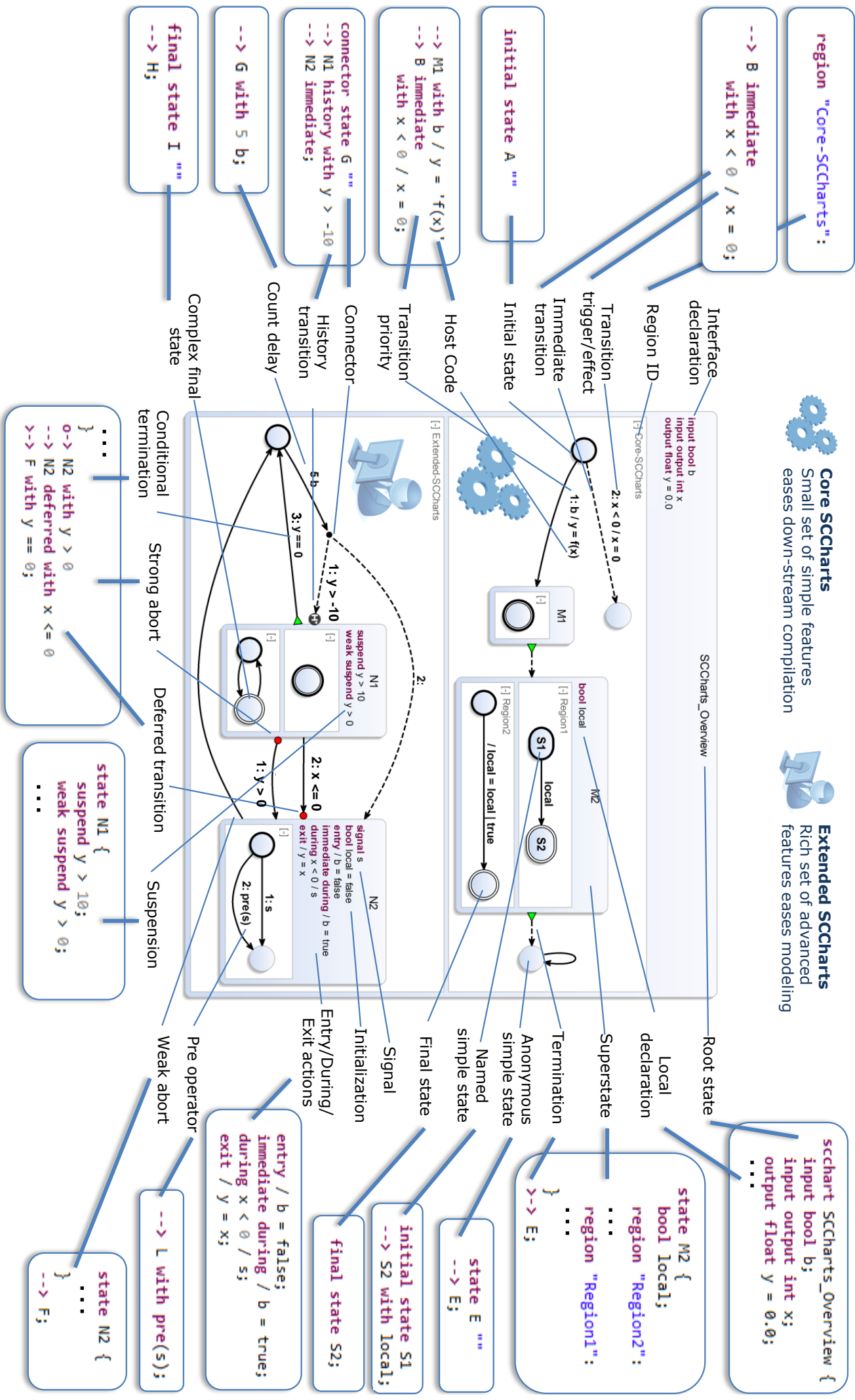


## **D. SCCharts Cheat Sheet**

Appendix D shows the SCCharts Cheat Sheet as it was presented during the Synchron Workshop in December 2016 in Bamberg. Throughout the whole development of SCCharts, this general view – sometimes with slight modifications – serves as an overview to all SCCharts features.

**Core SCCharts**  
Small set of simple features  
eases down-stream compilation

**Extended SCCharts**  
Rich set of advanced  
features eases modeling



# SCcharts Cheat Sheet

```

@diagram[KLayLayered] false
@HVLayout
scchart ABRO {
  input bool A, B, R;
  output bool O;
  region Main:

  initial state ABO "ABthenO" {
    entry / O = false;

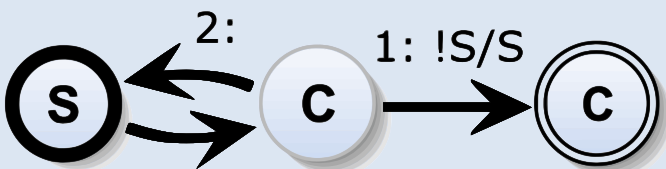
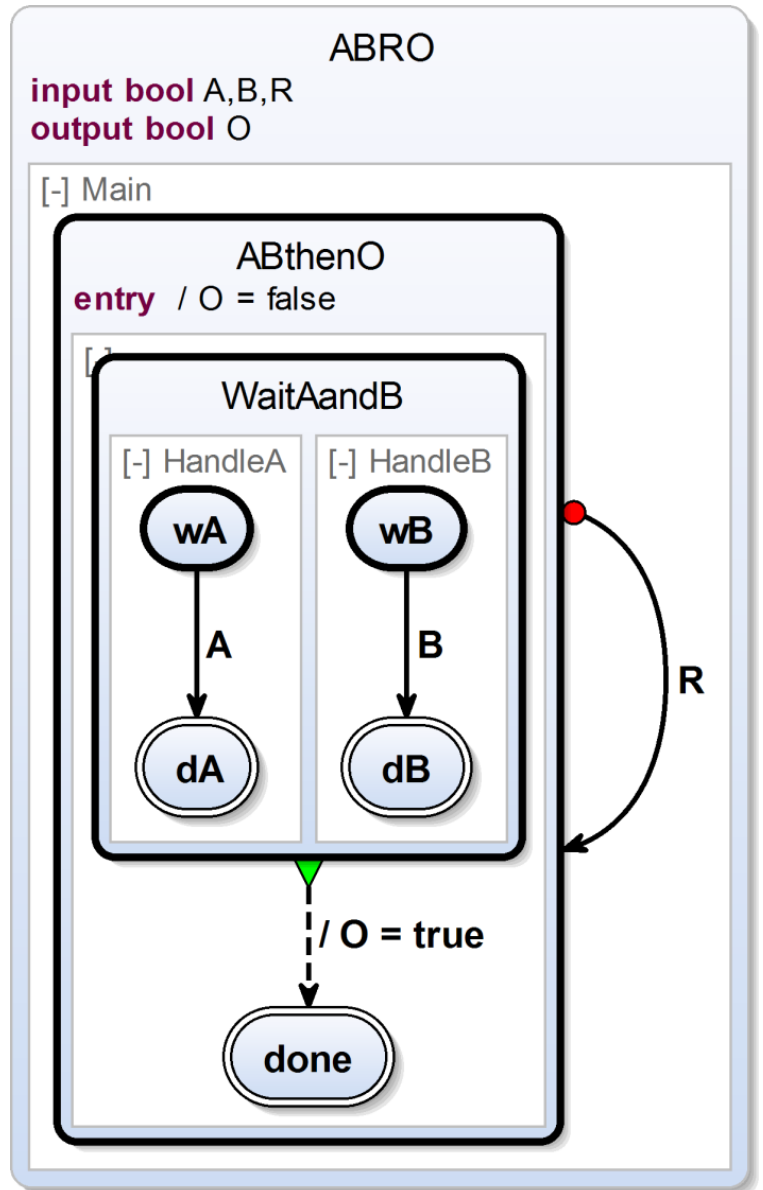
    initial state WaitAandB {

      region HandleA:
        initial state wA
        --> dA with A;
        final state dA;

      region HandleB:
        initial state wB
        --> dB with B;
        final state dB;
    }
    >-> done with / O = true;

    final state done;
  }
  o-> ABO with R;
}

```



# SCCharts Cheat Sheet