Debugging SCCharts

Lena Grimm

Bachelor Thesis
September 2016

Real-Time and Embedded Systems Group
Prof. Dr. Reinhard von Hanxleden
Department of Computer Science
Kiel University

Advised by
Dipl.-Inf. Ass. jur. Insa Fuhrmann

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel, den 29.09.2016

Abstract

Debugging a program can be a challenging task and there are many different strategies how this problem can be approached. For most languages there are sufficient features that facilitate finding the error and examining the state of the program at a certain time.

For the relatively young graphical modeling language Sequentially Constructive Charts (SCCharts) [HDM+14a], the editor support is still evolving. For the SCCharts editor in the Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER) project, there are already some features that may help finding the cause if there is a bug in a model. However there is no possibility that allows to stop execution when it reaches a specific model element. This would not only be useful but also correlates with the widely spread understanding of a debugger.

In this thesis possibilities that support bug finding or that prevent the occurrence of errors by providing additional features are discussed and evaluated. Not only a debugger itself can be helpful but also a readily accessible documentation helps deliver the semantics of model elements. Another strategy is discussed that involves static analysis of the SCCharts [HDM+14b]. This can help prevent errors from happening and it is further analyzed how this can be achieved. In the context of programming SCCharts for LEGO Mindstorm Robots another strategy is considered that involves saving the execution trace for a robot so that it can be reviewed after execution.

Finally a debugger is presented that uses the KIELER Execution Manager (KIEM) [MFH09] to provide options to stop execution according to breakpoints that can be inserted in the textual representation of SCCharts. A visual equivalent is added to the generated graphical SCChart. Correspondingly, a comfortable way of enabling and disabling debugging features is implemented and it is possible to fast forward the execution so that there is no need to manually step through the model until an interesting execution part is reached.

Contents

Contents vii

List of Figures ix

List of Tables xi

1 Introduction 1
- 1.1 KIELER . . . 2
  - 1.1.1 SCCharts . . . 2
  - 1.1.2 KIEM . . . 4
- 1.2 Problem Statement . . . 5
- 1.3 Contributions . . . 5
- 1.4 Outline . . . 6

2 Related Work 7
- 2.1 General Debugging . . . 7
  - 2.1.1 Java Development Tooling (JDT) Debug . . . 7
  - 2.1.2 Gnu Project Debugger (GDB) . . . 8
  - 2.1.3 SCADE Suite . . . 8
- 2.2 SCCharts Debugging . . . 9
  - 2.2.1 Compiler Selection . . . 9
  - 2.2.2 Dependency Analysis . . . 10

3 Exploration 13
- 3.1 Debugging in General . . . 13
  - 3.1.1 Definition . . . 13
  - 3.1.2 Stages . . . 16
- 3.2 Explored Approaches . . . 18
  - 3.2.1 Embedded Systems . . . 18
  - 3.2.2 Static Analysis . . . 18
  - 3.2.3 Execution Trace . . . 19
  - 3.2.4 Breakpoints . . . 21
- 3.3 Further Ideas . . . 21

4 Used Technologies 23
4.1 KIEM 23
4.1.1 Data Table 24
4.1.2 KART 25
4.1.3 Simulator 26
4.1.4 Synchronous Signal View 26
4.2 Eclipse SDK 26
4.2.1 Eclipse Platform 26
4.2.2 Plugin Developer Environment 28

5 The SCCharts Debugger 29
5.1 Simulation Framework 29
5.2 Elements 31
5.2.1 Text Ruler 31
5.2.2 Right-Click Menu 31
5.2.3 Breakpoints View 31
5.2.4 Breakpoint Visualization 32
5.2.5 Controls 33
5.2.6 Simulation Perspective 33
5.3 Usage 34
5.3.1 Adding a breakpoint 34
5.3.2 Creating an .eso-file 37
5.3.3 Schedules 39
5.3.4 Starting Execution 40

6 Implementation 43
6.1 Implementation Goals 43
6.2 Preliminaries in Editor 44
6.3 Preliminaries in Visualization 46
6.4 Simulation 47
6.5 Perspective 48

7 Evaluation 51
7.1 Limitations 51
7.2 Testing with a Bar Code Reader 53
7.2.1 Prerequisites and Problem Statement 53
7.2.2 SCCharts Model of Bar Code Reader 55
7.2.3 Possible Problems 55
7.2.4 Analysis of Usability 57

8 Conclusion **59**
8.1 Summary 59
8.2 Future Work 60
8.2.1 General Future Work 60
8.2.2 Optimizations 60

A Textual Description of Bar Code Reader **65**

B User Manual **73**
B.1 Requirements 73
B.2 KIELER SCCharts Debugger 74

C Acronyms **77**

List of Figures

1.1 Overview of the KIELER project 2
1.2 Dependencies of SCCharts transformation 3
1.3 Overview of SCCharts elements 4

2.1 KIELER compiler selection with the `Dependency` transformation chosen. . . 10
2.2 Visualization of different dependencies in a SCG. 11

3.1 A method in `C` that sums up all array elements 14
3.2 Expample for fixing the root cause in an SCChart 15
3.3 Iterative debugging Process presented by Butcher [BC09] 16
3.4 Problem with instantaneous loops in SCCharts 20
3.5 Binary decision diagrams representing the formula $a \wedge (b \vee c)$ 22

4.1 Overview of the elements that belong to KIEM highlighted in red rectangles 25
4.2 Overview of the Eclipse Platform Architecture 27

5.1 Screenshot of KIELER debugger elements 30
5.2 KIELER Modeling Perspective 32
5.3 Visualization of breakpoints in SCCharts 33
5.4 Synchronous Signals View 34
5.5 Semantics of a state breakpoint 36
5.6 Semantics of a state breakpoint with hierarchical states 37
5.7 Semantics of a transition breakpoint 38
5.8 Example of an .eso file with four different traces 39
5.9 KIELER Automated Regression Testing (KART) configuration menu . . . 39
5.10 Pop-up dialog shown when no .eso-file is provided 40
5.11 Example of a schedule menu 40

6.1 Extension Points grouped by its function 44
6.2 Method `delegateToggleToTarget()` of class `TextRulerHandler` 45
6.3 Extract of the `step(...)` method 49

7.1 Problems of SCCharts breakpoints with immediate transitions 52
7.2 Structure of a bar code 53
7.3 SCCharts model of the bar code reader 54

7.4 First lines of the created .eso-file for the bar code reader 56
8.1 Synchronous Java (SJ) Instruction View 63
B.1 Summary of the button functionalities of the debugger 75

List of Tables

3.1 Errors categorized by cause, moment of occurrence and the respective unit as presented by Halang [HK99] 14

6.1 Methods implemented in `BreakpointListener` 46
6.2 Significant Methods in `DataComponent` 48

Chapter 1

Introduction

Looking at airplanes, cars, trains, also at the smart phone laying on a table or at the coffee machine brewing some coffee, all these have one thing in common — they contain embedded systems.

An embedded system is a kind of computer system that is integrated into a real world context but unlike a personal computer it needs to operate according to inputs its environment provides in real-time. These inputs might come from a sensor for an airplane or the power button of a coffee machine. The reactions thereafter need to be functionally correct and on time. Additionally, the system has to achieve this while running for a long time without having the option to restart it on a regular basis.

Consequently, this requires the software for embedded systems to be robust. In order to meet the time specifications, functional concurrency is often necessary. To express concurrency in languages like `C` or `Java` often presents problems, because it is associated with a non-determinate behavior due to the occurrence of race conditions. Especially for safety-critical applications, determinacy is indispensable. Hansen [Han99] and Lee [Lee06] both already analyzed the problem with parallelism in such languages and the consequences resulting from the use of threads.

A computation is considered determinate if the same sequence of inputs produces the same sequence of outputs. Thus, it must be possible to abstract from a system so its behavior can be analyzed and proven to work correctly. Otherwise, if the output differs with identical input it would never be predictable what output will be produced. To overcome this problem, synchronous languages have been introduced.

Synchronous languages such as Esterel [Ber00], Signal [GGB+91] or Lustre [HCR+91] are often used when designing a reactive system. They offer a determinate behavior and enable timing analysis so the problem stated above can be resolved [BCE+03]. Nevertheless synchronous languages also come with many restrictions. Some programming techniques that might be used in other programming languages would cause a similar program in a synchronous language to be rejected. For example, only a single value is allowed for an output in one cycle of reading inputs and computing outputs.

For this reason the Sequentially Constructive Model of Computation (SC MoC) [HMA+13c] was introduced. This model serves as a conservative extension to the synchronous Model of Computation (MoC) [AMH+14] and enables multiple writes to signals,

as long as the final value at the end of a tick is assured to always be the same. The concurrent variable access is ordered by the initialize-update-read protocol [HMA+13a; HMA+13b] and thus a more intuitive style of programming can be allowed and additionally deterministic use of concurrency is guaranteed.

1.1 KIELER

The Kiel Integrated Environment for Layout Eclipse Rich Client [HFS11] is a research project developed by the Real-Time and Embedded Systems group at Kiel University. It is used to provide modeling software for SCCharts, amongst others, and aims to improve the model-based design of complex systems. In addition to the editor that is used to describe SCCharts in a textual way, there is a visual component that is automatically updated accordingly so it shows the visual representation of the textual description. This automatic layout generation frees the user from drawing models by hand, which is very time consuming, especially as models grow.

The overall project can be structured into three main parts: layout, pragmatics and semantics. In Figure 1.1 this separation is represented visually. In this thesis the focus is on the semantics part regarding SCCharts and the execution Framework KIEM.

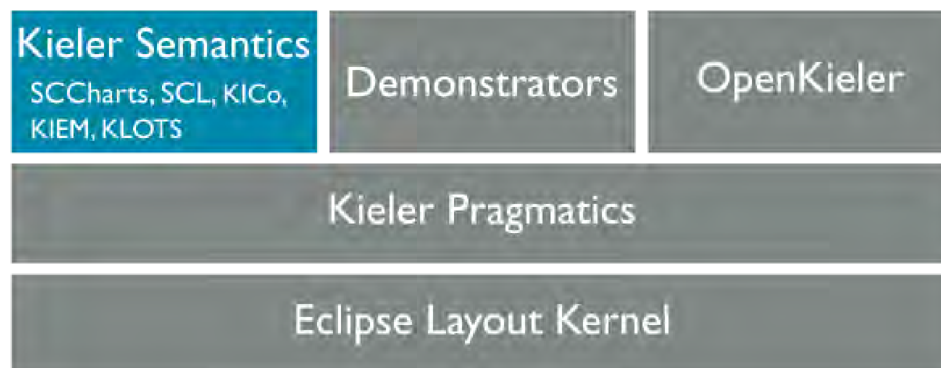


Figure 1.1. Modified overview of the KIELER project as presented in the documentation[1]

1.1.1 SCCharts

SCCharts [HDM+14a] are a state-chart modeling language defined on the basis of the SC MoC. They provide a combination of features which were initially introduced as part of SyncCharts [And95] or Statecharts [Har87]. Also some elements from Quartz [Sch09],

[1] http://rtsys.informatik.uni-kiel.de/kieler

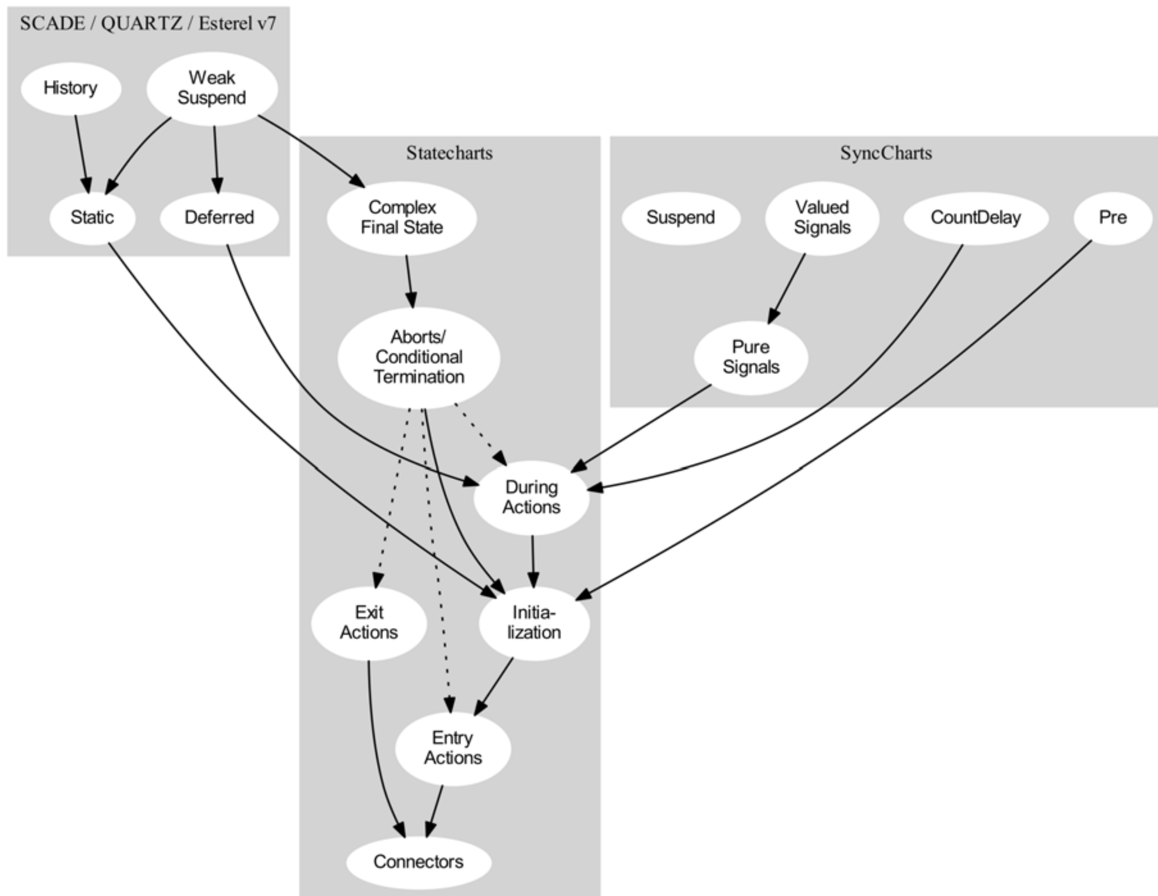


Figure 1.2. Dependencies of the SCCharts transformation and their origin [HDM+14a]

SCADE [06] and Esterel [Ber00] are included and can be used for modeling. Figure 1.2 gives an overview of the different transformation options and where they originated. There is a textual and a visual syntax defined for describing SCCharts and the visual notation is closely related to the SyncCharts notation.

The semantics of SCCharts are defined on a determinate basis, but the safety-critical focus is also reflected in the definition of the language. Figure 1.3 shows the division between elements of Core SCCharts and those that are extended features. All Extended SCCharts can be transformed into Core SCCharts. Therefore formal analysis and verification is facilitated as much as possible.

Different projects have been carried out using SCCharts like modeling software for the LEGO Mindstorm Robot[2] or a model railway controller [The06]. Moreover, hardware

[2] http://www.lego.com/de-de/mindstorms

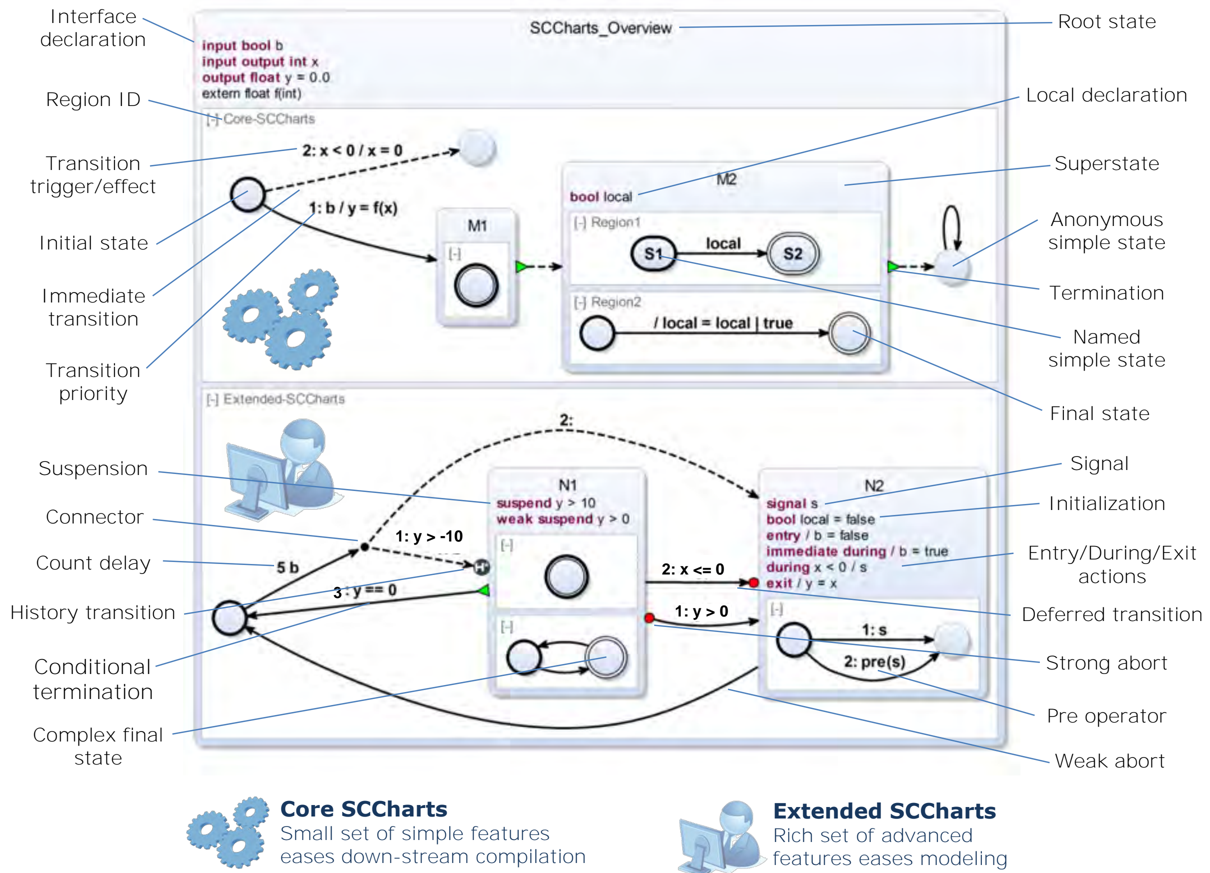


Figure 1.3. Overview of SCCharts elements whereas the upper region consists of core features and the lower region expresses the extended features [Mot16]

code like `VHDL` can be generated and programming an Arduino[3] or similar architectures is possible through modeling its behavior in a SCChart.

1.1.2 KIEM

The KIELER Execution Manager (KIEM) is a simulation framework for different types of models such as SCCharts and was introduced by Motika [MFH09]. It provides the option to chose a simulation background and connects the simulation with visual representations. For the simulation itself different backgrounds can be used. The The simulation is thus for example based on `C` Code. Other components can also be added in a generic way and loaded or deleted so the user can use the components they like and removes the ones they do not need.

[3] `https://www.arduino.cc/`

1.2 Problem Statement

The process of finding errors in SCCharts can become an elaborate task with growing models and a more complicated control flow. For development reasons, there had been several surveys regarding features and usage of the SCCharts. Throughout these the request for more debugging possibilities was paramount. Also, Smyth et al. [SMS+15] stated in their technical report about the railway project that finding the cause for bugs resulted in enormous time consumption. Additionally, the survey collected after the embedded systems lecture held 2015/16 stated that the cause for functional or compilation errors of the models could hardly be identified due to a lack of debugging tools.

There are many different ways to approach this problem since debugging can be viewed from many different perspectives. Several possibilities can be considered but they might not be a good solution in all cases and their benefit and usability differs greatly.

In general a debugging feature should facilitate finding errors in programs. This can result in some tool that helps to understand the control flow or just in comfortably provided information about model elements that help finding the error. Examples for this feature are documentations of model semantics embedded in the software or suggestions on error solution due to compilation errors.

1.3 Contributions

The ideas presented below are all evaluated in the context of debugging SCCharts in KIELER. One part of the thesis is the exploration of possible options for enabling additional features. For every option it is analyzed under which circumstances they would be most useful. Additionally, it is explored if it is possible to implement them in the context of KIELER, and if there already exist features that fit into one of these options. In the latter case it is further analyzed if the features work and if they can even be advanced.

The second part of this thesis is the implementation of a debugger in KIELER. During implementation particular attention is put on designing it as intuitively as possible. In general, there are two main parts of this implementation, the visualization of needed features and their functionality. Both parts have certain constraints that they need to fulfill.

The visualization of the debugger aims to have a familiar looking Eclipse appearance. Elements that still exist in general Eclipse applications should be adopted and the meaning of certain icons is be maintained as far as possible. For newly introduced icons they are oriented to be self-explanatory.

For the functionalities of the debugger there are also some aspects that are essential. On the one hand execution needs to stop at the moment a model part with an active

breakpoint is reached and afterwards the execution can be resumed. Fast forwarding the execution is mandatory so in case there is no breakpoint in a model section it is executed with as little time delay as possible.

1.4 Outline

The remainder of this thesis is organized as follows.

Chapter 2 presents related work. This includes debugger strategies in other programming languages and the given possibilities in other modeling tools regarding debugging. Also general discussions are referenced here that contain information about what can be defined as a debugging feature.

In Chapter 3 the possibilities for debugging in the context of SCCharts in KIELER is discussed and evaluated. Their feasibility can be estimated and thus good debugging features can be extracted.

Chapter 4 specifies the used technologies and more details on their functionality.

Chapter 5 presents the debugger in KIELER and its functions. New elements are introduced and explained and the general work flow strategy is presented. Besides, the semantics of breakpoints in different contexts is illustrated.

The implementation of the debugger is explained in more detail in Chapter 6. The relation to the given KIELER elements is discussed and it is further elaborated in which way the debugger is embedded.

In consequence, Chapter 7 evaluates the usability of the implemented debugger and results on debugging even larger models is presented.

Finally the presented solutions are summarized in Chapter 8 and future work suggestions are made.

Appendix A holds the source code for the bar code reader presented in Chapter 7.

Appendix B contains a User Manual. At this point the focus is on the functions provided to the user and implementation details are hidden.

Chapter 2

Related Work

Debugging is a vast field. Multiple different approaches have been introduced in literature and industry. In the following, some of these approaches in development tools are introduced. Section 2.1 offers more detail on debugging in general. This includes Eclipse debugging features and special debuggers for certain languages. Section 2.2 introduces debugging-related tools in KIELER that were already implemented before this thesis and explains the range of their assistance in debugging.

2.1 General Debugging

Debugging has always been one of the most costly parts of software development [XY03; Ves85]. A lot of time is wasted while searching for the cause of a bug [PO11]. Techniques for imperative languages like `C` and `Java` are well elaborated and many different approaches have been leveraged [KY03; OS02; CKL04]. For embedded systems, a debugger is even more complex. Concurrent regions and non-repeatability make it hard to understand the program's behavior. MCDowell and Hembold [MH89] examined the problems of debugging concurrent programs. They introduced different approaches for general parallel programming.

In this section, general debugging approaches are presented. The Java Development Tooling (JDT) is embedded in Eclipse and offers tools in combination with `Java`. Furthermore, the GDB is presented with a focus on `C` code debugging. In the end of the section, the SCADE Suite and its offered features are introduced.

2.1.1 JDT Debug

The Java Development Tooling (JDT)[1] project is a plug-in collection developed for Eclipse. It includes tools for modeling `Java` applications in an Eclipse environment and is split into five main projects that modularize the overall project. One part of these five main components is JDT Debug. Debug support for Java applications and plug-in development is offered and this support is embedded in the Eclipse context.

[1] http://www.eclipse.org/jdt/

The Java Platform Debugger Architecture (JPDA) offers interfaces to be used in development environments. The Java debugging support works with any JPDA-compliant target Java Virtual Machine (VM). This VM can be launched in either run or debug mode. In debug mode changes in classes are loaded dynamically to the current execution as long as the Java VM still supports the changes.

This debugger implements principles closely related to the goals set for the KIELER debugger. It builds a debug model in the context of Eclipse and its functionalities are helpful for debugging in combination with programming Java in Eclipse. The difference between the debugger in this thesis and JDT debug is the programming language itself. The KIELER debugger works on the modeling language SCCharts. Additionally, reactive systems are modeled, therefore the environment must be emulated, too. This is not needed for the JDT debugger.

2.1.2 GDB

The Gnu Project Debugger (GDB)[2] is a debugger for languages like Ada, C, C++ amongst others. It was developed by the GNU-Project and many features are provided in order to find bugs.

One of the core functions is the specification of breakpoints. The execution halts at the moment a breakpoint is reached and the user can examine how the system state is at that moment. More code can be executed afterward line by line. Also they can change internal variables and examine the effects in the ongoing execution.

The purpose of such a kind of debugger is to see what is happening inside the executing program. This is the same approach that is implemented in this thesis. Also the specification of breakpoints is adapted.

Additionally, the SCCharts code can be compiled to C code. The main KIEM simulation component operates on this compiled code in C language. This offers the possibility to integrate GDB features into the KIEM simulation, but this option is out of the scope of this thesis.

2.1.3 SCADE Suite

The SCADE Suite is a tool published by Esterel Technologies[3]. It is used to design safety-critical software based on the data-flow oriented programming language Lustre [HCR+91]. Lustre is a synchronous language like SCCharts. Therefore, the debugging features implemented in SCADE can be realized similarly in the context of KIELER.

[2] https://www.gnu.org/s/gdb/

[3] http://www.esterel-technologies.com/

The simulation of models created in SCADE Suite is based on the generated code. Different traces can be recorded and replayed. Automatic testing is enabled and the definition of stop conditions and the setting of breakpoints is possible. Also variables and other data can be examined and tracked.

Some of these functionalities already exist in KIELER but especially the breakpoints are adapted for the context of SCCharts. In SCADE Suite, breakpoints can be specified on control, data and time criteria. The control approach is realized in the SCCharts debugger by allowing breakpoints at states and at transitions. The data breakpoints could be realized by extending the debugger to breakpoints on variables and thus changes on them. The time criteria breakpoint would be a nice extension to prevent infinite loops. The control flow approach is implemented because a lot of ideas can be realized through its use and additionally, this is also the main idea of a debugger for languages like `C` or `Java`. The time criteria and data breakpoints are not implemented in the scope of this thesis.

2.2 SCCharts Debugging

This section presents some elements that are already included in KIELER. They can be used to find errors in a designed model and help understand and visualize conceptual problems. Especially scheduling problems are often hard to understand for unexperienced users.

2.2.1 Compiler Selection

The KIELER compiler selection is a view that is placed at the bottom in the KIELER modeling perspective. The root SCCharts model is transformed to code but all steps towards the code generation can be selected manually. This enables the reviewing of model transformations.

In Figure 2.1 the different nodes of the compiler selection are shown. The internal transformations of the SCGraph node are opened and the Dependency transformation is chosen. This transformation analyzes existing dependencies in between concurrent threads in the Sequentially Constructive Graph (SCG) and visualizes them. The diagram view is adjusted accordingly with the chosen transformation criteria.

On the SCCharts level, the goal is to transform the model to Core SCCharts format. Core SCCharts do not include as many features and build an equivalent representation of the complex model in core elements. The semantics and the complexity of special constructs used in the model might become clearer. The next step is the Sequentially Constructive Graph. A Sequentially Constructive Graph (SCG) is a graph that shows the control flow of the corresponding SCChart and is then used to determine a schedule for

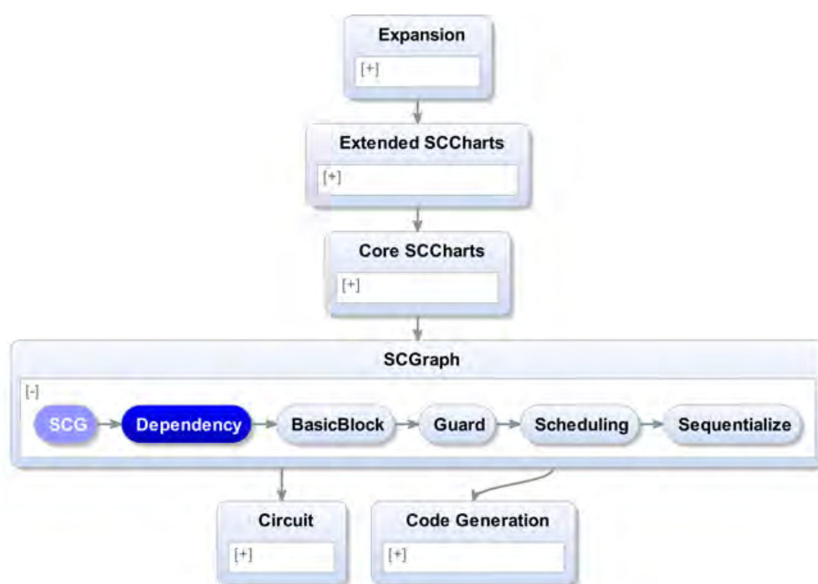


Figure 2.1. KIELER compiler selection with the Dependency transformation chosen.

the model. Problems relating to control flow and dependencies are possibly identified at this stage. In general the model is checked for constructiveness.

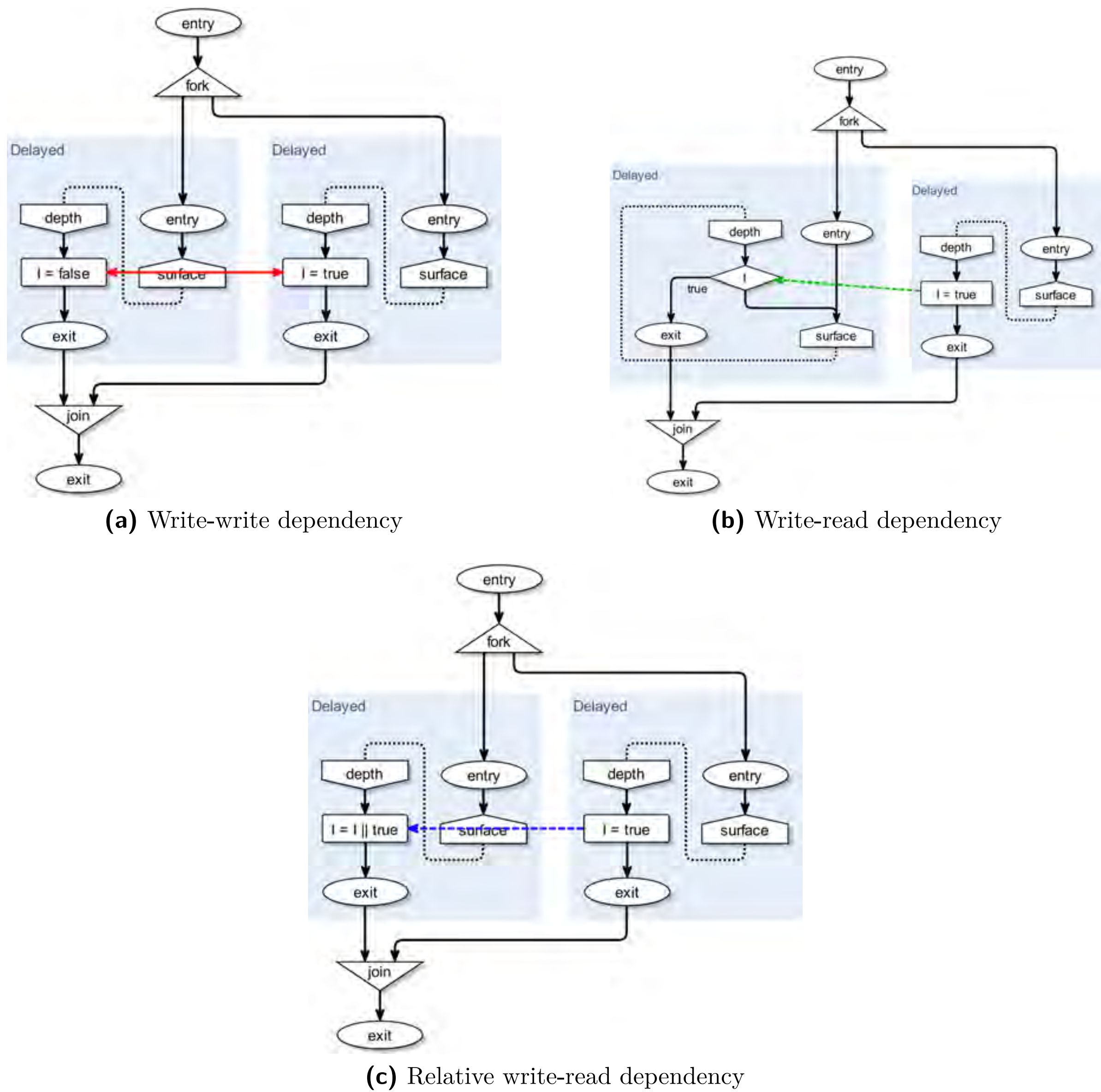
Lastly, the code is generated. This code can also be reviewed and checked but every stage is more abstract in comparison with the original model.

2.2.2 Dependency Analysis

Programming with synchronous languages might feel odd for users that are used to program with threads. Variable access is controlled and models with indeterminate variable access that would result in race conditions are rejected. Therefore conflicts in variable access may happen and result in a model that can not be scheduled due to non-constructiveness. Smyth analysed these variable dependencies in the corresponding SCG [Smy13].

Extending the SCGraph node in the KIELER compiler selection reveals the option Dependency. After this transformation is chosen, the corresponding SCChart is converted to an SCG and internal dependencies are visualized with arrows in different colors. In Figure 2.1, the compiler selection with the Dependency transformation is shown.

There are three different types of dependencies that are visualized. There are write-write dependencies, relative write-read dependencies and write-read dependencies. A relative write is an update of the old value. The new value is related to the old value. All these dependencies have consequences on the schedule created because they require certain operations to be executed in a defined order. Hence, it is possible that the model



(a) Write-write dependency

(b) Write-read dependency

(c) Relative write-read dependency

Figure 2.2. Visualization of different dependencies in a SCG.

becomes non-schedulable. Understanding the dependencies in the model helps understand scheduling problems. Therefore it is reasonable that this feature helps finding errors and bugs in the corresponding model. In Figure 2.2 the different visualizations of the dependencies are shown.

Chapter 3

Exploration

This chapter analyzes approaches for debugging SCCharts. Section 3.1 gives a general overview of debugging. The term is defined, and deeper developing processes are presented. Section 3.2 contains a differentiation between debugging software for general computer application and embedded systems. Already developed debugging approaches for reactive systems are evaluated in the context of SCCharts. The last section sketches further debugging strategies and ideas for implementation in the KIELER context.

3.1 Debugging in General

This section presents the definition for debugging and describes steps that are linked to it. The act of debugging is often referred to in the context of just fixing a bug, but there is much more to debugging. The main goal is to find the root cause for the bug and understand why the error could occur. As a consequence this error should be prevented from happening again.

3.1.1 Definition

A vast amount of debugging tools are available that help developers find the cause for bugs. Many of these tools have in common that they are limited to software errors. Software errors are just a fraction of the different possible causes for a bug. Table 3.1 categorizes these different errors with the software errors marked in bold red. Those are the errors that this thesis will focus on, but it is important to keep in mind that there are still many more possible causes.

In general, a system has a bug, when its actual behavior does not match the expected one. The cause for this unexpected behavior can be a faulty input or wrong modeling. Identifying the origin for the unexpected behavior is the main goal of debugging. Often, the fault for a bug is not identified, but the system is changed to avoid the error. This may fix the bug, but it does not debug the system. The programmer did not understand the main cause for the error and therefore, they did not fix it but simply made the system work. In the following, an example for a method in `C` is given.

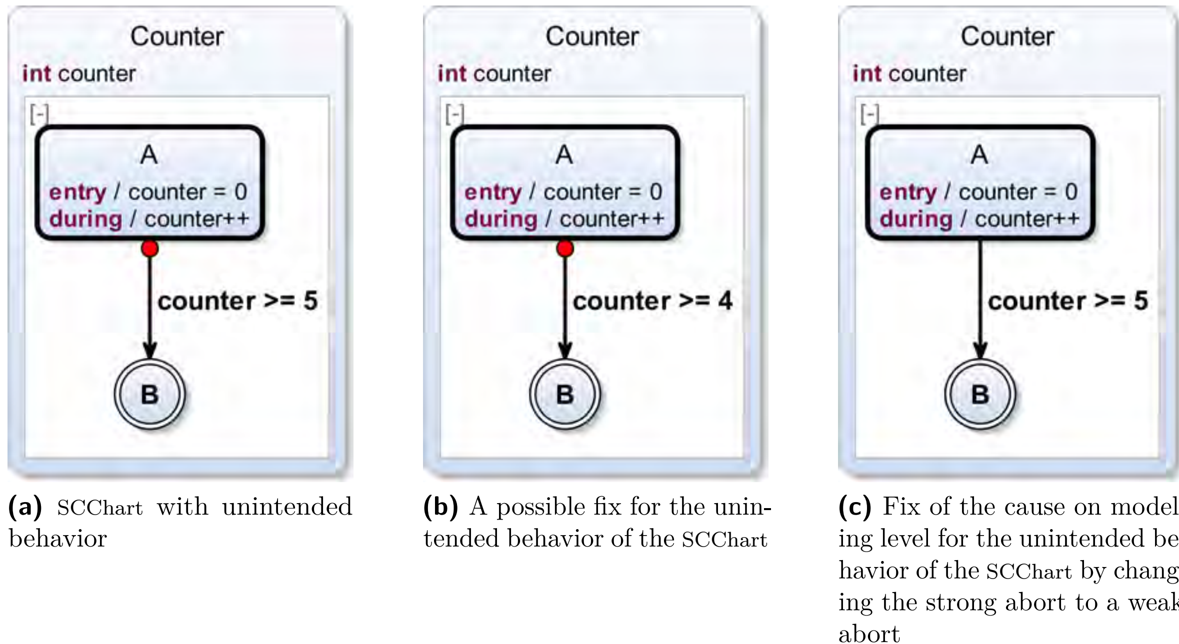
Table 3.1. Errors categorized by cause, moment of occurrence and the respective unit as presented by Halang [HK99]

Cause for the error	Moment of occurrence	Faulty Unit	Examples
Physical and chemical Processes	Outside of Usage	Hardware	Material Manufacturing Error
	Within Usage	Hardware	Interfering Signals Device Failure
Human	Outside of Usage	Hardware	Construction Error Manufacturing Error Maintenance Error
		Software	**Software Design Error** **Programming Error** **Configuration Error**
	Within Usage	None	Operating Error Configuration Error

Figure 3.1 shows a function in the language `C`. It takes an array of integer and sums them up. The size of the array is also passed as a parameter. This function can be called from outside. For some reason, the `size` variable for the array length is one too small. Adding the line `size++;` prior to the for-loop fixes the symptoms of the bug. However, this function can also be called from positions where the size is not one off and this would result in an overflow. Therefore, the bug was not identified but covered. Locating and fixing the position where the function is called with a wrong size is the desired approach for debugging.

```
1  int sum_up(int array[], int size)
2  {
3      int i;
4      int sum = 0;
5
6      for (i = 0; i < size; i++) {
7          sum += array[i];
8      }
9
10     return sum;
11 }
```

Figure 3.1. A method in `C` that sums up all array elements



(a) SCChart with unintended behavior

(b) A possible fix for the unintended behavior of the SCChart

(c) Fix of the cause on modeling level for the unintended behavior of the SCChart by changing the strong abort to a weak abort

Figure 3.2. Expample for fixing the root cause in an SCChart

In the context of SCCharts a fix for a bug can also be evaluated in its modeling approach. Figure 3.2(a) shows an SCChart. It is supposed to count five ticks and then, the final state should be entered. However, this SCChart enters the final state one tick too late.

This problem can be fixed by changing the trigger of the transition to counter $\geqslant$ 4. Figure 3.2(b) visualizes this SCChart. This will cause the model to enter state B in the fifth tick and fix the bug. The model meets its specification but apparently the user did not understand where the cause for the bug arises.

Additionally, the counter does not work as a counter. The moment when the final state B is entered, it still holds the value 4. This is not an error. The counter variable is not visible outside, but it does not implement the intuitive meaning for a counter which would be 5 for the specification of the model.

The strong abort transition is the reason that the final state is entered one tick too late. Its trigger is not checked directly after the counter is increased. Only in the beginning of the tick, the trigger is checked. Changing the strong abort to a weak abort solves the problem. This approach for fixing the bug works with the provided model elements and does not create a work around. Figure 3.2(c) shows this model. The counter now holds the value 5 when entering state B.

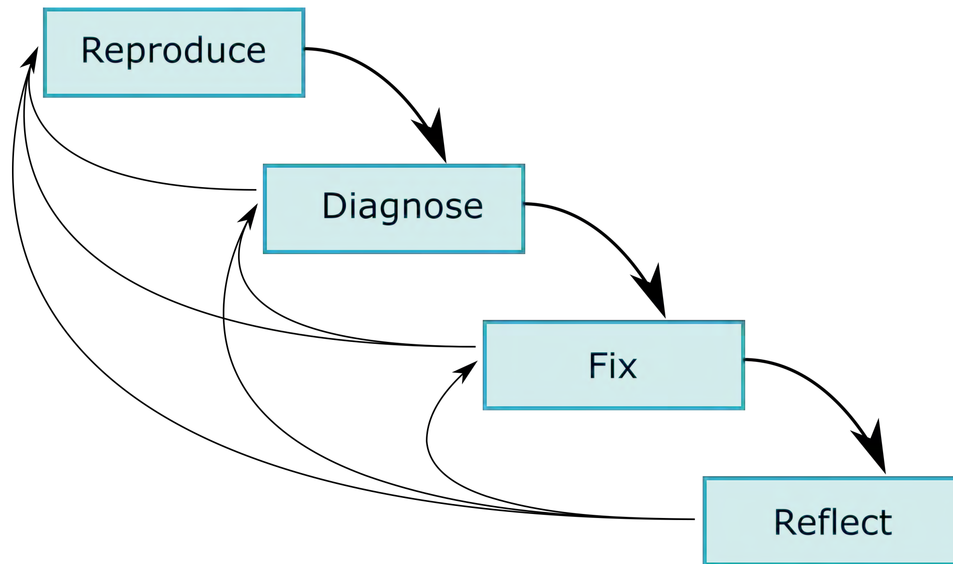


Figure 3.3. Iterative debugging Process presented by Butcher [BC09]

Butcher [BC09] introduced different goals for debugging. The bug should be fixed at its root and not at a higher level that only causes the bug to be covered. Also the programmer must be careful not to break anything else when trying to debug. At last, the knowledge gained should be used to ensure that the same kind of problem does not occur at a similar situation in the program.

3.1.2 Stages

The core debugging process goes through four different stages as defined by Butcher [BC09]. Each stage should be completed before starting in the next stage. In Figure 3.3 these four stages are shown and the development work flow is visualized through arrows. During this entire iterative process, it is important to know what output is expected. A behavior can not be wrong if there is no clear definition of what would be the right behavior. These specifications must be set and defined precisely. Lee and Seshia already paraphrased Young by saying that a system with no specification can not behave right or wrong, it is only surprising [LS11]. This summarizes the need for a good model specification. In the following, each stage of the debugging process and its focus are discussed in more detail.

Reproduce The first step towards understanding a bug is the ability to reproduce it. In case it is not possible to reproduce the bug, it is not possible to make any reasonable

debug progress. The cause for the bug cannot be proven to be right and also, in case of a fix, the absence of the bug can not be demonstrated.

Diagnose This the where debugging happens. The root cause must be identified and isolated. It is possible that an error has more than one cause, but in many cases it does not, so assuming only one cause is best practice for the start.

It is important to keep track of the changed things. This gives the possibility to backtrack changes made in case something breaks because of them. Often, many different code parts are worked on and it is easy to loose track. Debugger help to investigate internal system states and thus to understand why a certain system state is reached and the effects of changed code. There are mainly three different reasons why a debugger is used. Those are listed in the following.

(1) It is used to check that an implementation works correctly by making single steps.

(2) There can be a set-up theory about why the system behaves the way it does and it is supposed to be verified.

(3) The programmer does not understand why the code behaves in a certain way.

An error might be more or less complicated and many tools exist that try to help the programmer, but in the end, it is the intellect that has to do the work. Therefore, understanding the system is the first and most important achievement on the way of debugging. The goal is to find the cause for the bug, not only its symptoms.

Fix The diagnosis is finished and the cause for the error is found. In many cases this already makes up the most difficult part of the debugging process. However, the bug must still be fixed without breaking anything else. At this point a structured and disciplined approach is required. The code should be maintainable and of high quality. If tests exist, they should be run after the fixing is done to ensure correct behavior. Furthermore, it is good practice to add a test, verifying the behavior accomplished after the fixing of the bug.

Reflect Debugging also brings some rework with it. It is important to reflect on the whole error. Most likely, the error did not occur in every case, and also the moments where everything went right are just as important to understand. This is especially relevant when thinking about the fix. Situations that did not cause a bug should not cause one after the fix.

Additionally, as mentioned in the paragraph above, it should be ensured that the bug does not happen again. This may include automatic testing or the improvement

of maybe misleading interfaces or documentation. Lastly, the programmer should think about similar situations, where the same kind of bug might evolve. Those should also be fixed.

3.2 Explored Approaches

This section introduces debugging strategies in the context of embedded systems. The strategy of a debugger as introduced later in Chapter 5 is classified and further ideas are presented.

3.2.1 Embedded Systems

Debugging embedded systems can be tedious. They typically run on hardware in a specific environment that is different from the developing environment. Therefore gaining access to the system state and the information required is already elaborate. Also, the process of debugging has an influence on the time behavior and therefore, debugging in the real-time context is especially hard. Different tools with different focuses exist that fit into the context of debugging embedded systems.

There are remote debugging tools that require the embedded system to be connected to the development system. The debugger runs on the development system and controls the embedded one. The advantage of this approach is that the model is actually running on the end target system.

The next approach are in-circuit emulators. In general an in-circuit emulator is a debugger that connects hardware and software debugging and is able to provide detailed access to the internals of the system. One of the frequently used interfaces for this use is JTAG.

The following is a simplified variant of this approach. A second target system is built for the developing process. It includes additional interfaces and support for other test facilities. Often support for in-circuit emulators is already given.

The last approach is the emulation. This is the approach used through the KIELER KIEM debugger. The model is simulated on the development machine and this saves a great amount of time. The procedure of editing, building and testing is shortened in contrast to deploying it on the embedded system for every test execution. A lot of insight in the model is given, however, the model is just simulated. In the actual embedded context, errors might occur that are caused by unpredictable inputs that can hardly occur in the simulation environment and the specifics of the underlying architecture.

3.2.2 Static Analysis

Many debugging approaches, including the debugger implemented in this thesis, rely on dynamic analysis. The code is reviewed as it executes or during simulation. A disadvantage of the dynamic debugging is that the model must be at least compilable. Non-compilable models cannot be debugged with the implemented debugger.

However, Butcher [BC09] stated that many problems can be detected using a static analysis like the detection of dead code in means of unreachable code. An analysis of reachable and, in consequence, unreachable states is already implemented in KIELER. This analysis could be extended. Unused variables could be marked. Also, unreachable transitions can be highlighted that cannot be taken because of higher priorities on other outgoing transitions whose trigger is always evaluated to true.

Additionally, a dependency analysis [Smy13] is already implemented. It helps to understand problems in variable access in concurrent regions. This brings helpful information about the underlying MoC to the modeling process. The SC MoC might be unfamiliar to new users and therefore this information could help the user understand this MoC.

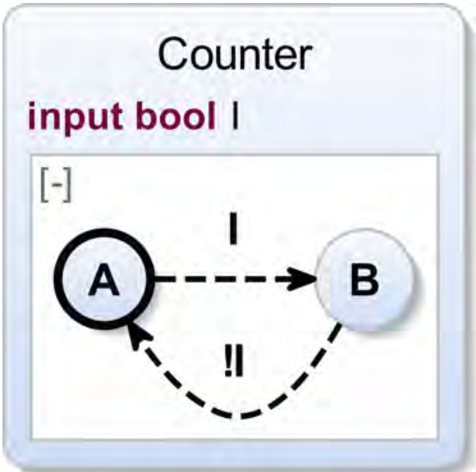
Another possibility for static analysis is the detection of instantaneous loops. Figure 3.4(a) shows an SCChart with a potentially instantaneous loop. The input I cannot be true and false in the same tick, but still this model is not schedulable. Here, static analysis is used to provide more flexibility to the compiler and not help the user fix errors.

However, there is a highlighting in the SCG that shows the potentially instantaneous elements. Figure 3.4 illustrates the corresponding SCG with the highlighted blocks that form a problem for the scheduling path.

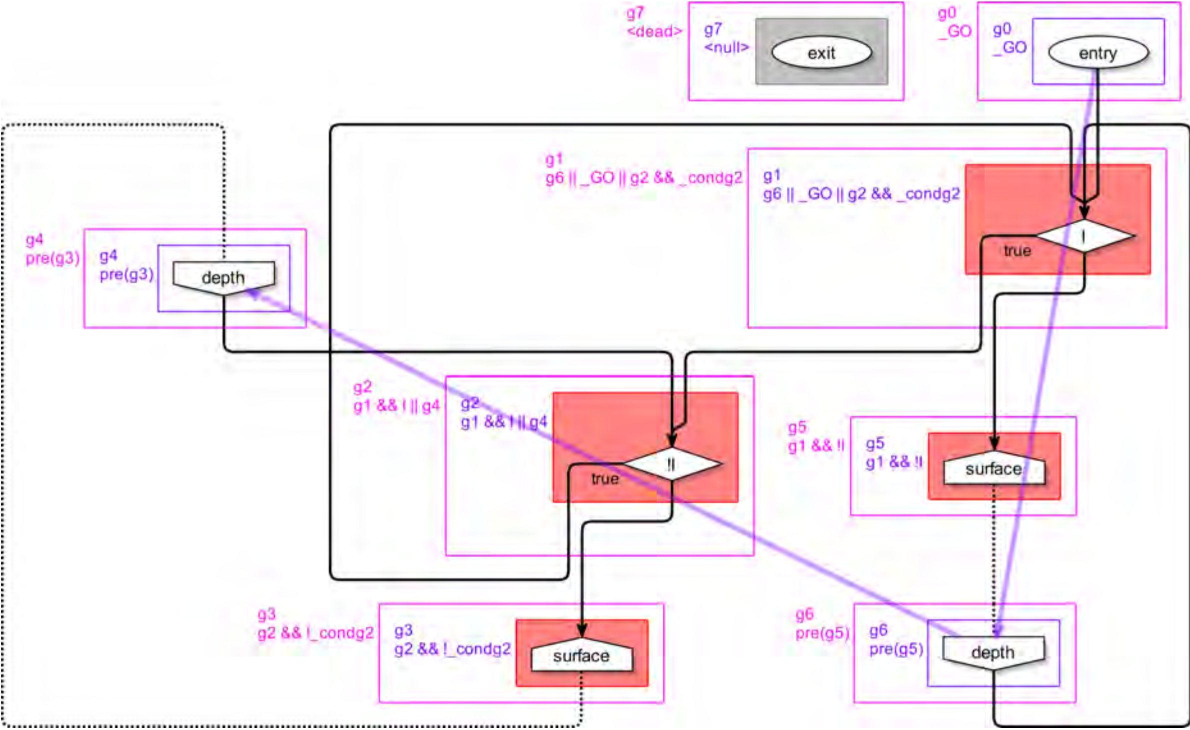
3.2.3 Execution Trace

For this approach, different options can be taken. In general, the system state is supposed to be logged in order to be able to review it. This is the most accurate debugging, because it runs on the actual system without interference of the development system. The trace of the system states must then be reviewed in order to find the error. This is also a form of dynamic debugging.

Quade [QM12] listed tracing mechanisms as a possibility to review internal states. The state of the system is recorded in a file or the system-specific `print(...)` method is used. It is recommended to add a time-stamp, the name of the file and the line number to be able to backtrack the trace if using `print(...)` statements. This enables to locate which statements have been reached by the execution and it is especially helpful when talking about concurrent systems. The execution order of different threads can be put together. In the embedded context the `print(...)` method usually does not exist. An



(a) SCChart with an instantaneous loop



(b) SCG with highlighted block that cause a problem in the scheduling path

Figure 3.4. Problem with instantaneous loops in SCCharts

LED can be used that informs about internal variables and states by changing its color or getting switched on or off.

For the context of SCCharts, the saving of execution traces must be handled differently for the specific devices. For Lego Mindstorm, the `print(...)` method saves the output in an internal .log-file. This allows to use it in means of saving the trace. This process could be integrated in the code generation for Mindstorms. Note that the `print(...)` method takes a lot of time executing on the Mindstorm. It is possible that activating and deactivating these statements causes the model to react differently in terms of timing.

3.2.4 Breakpoints

The breakpoint approach is most likely added to the emulation. A constraint can be defined that causes the simulation to pause the moment it evaluates to true. These constraints can be defined on control, time or data.

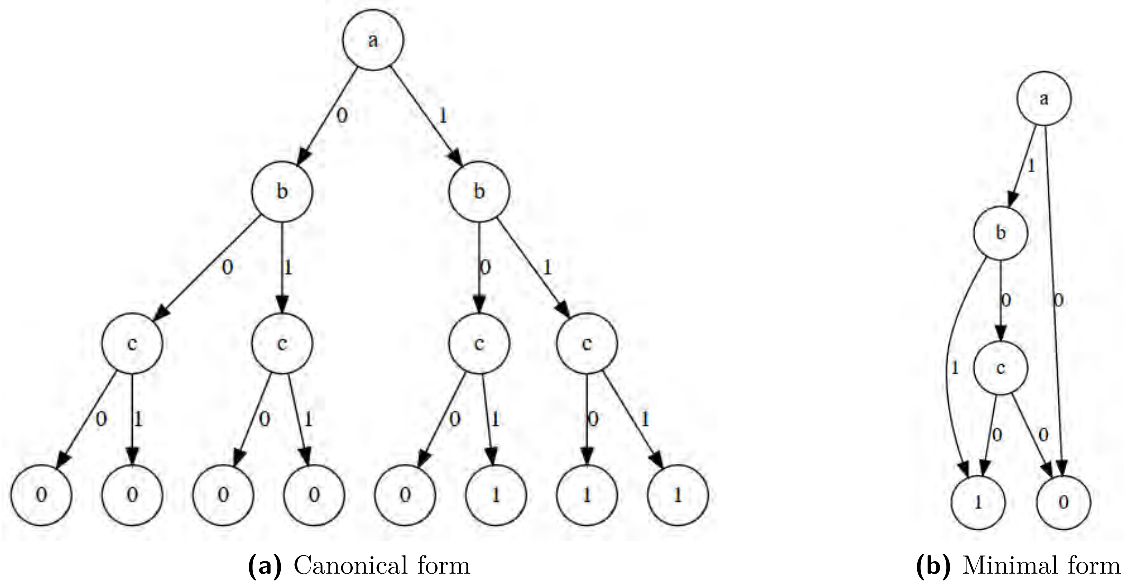
A control constraint specifies a moment in the code and when reaching it, simulation is paused. The time constraint causes the simulation to be paused after a specified time. At last, the data constraint can be specified on variables. On access or change the simulation pauses. The control variant is implemented through the KIELER debugger and is further described in Chapter 5 and Chapter 6.

3.3 Further Ideas

Especially for embedded system, there is another approach to ensure a certain amount of correctness. This approach has been recommended and elaborated by many authors, Lee and Kroeger amongst others [LS11; KM08].

The main idea is to formulate invariants from the model specification. The behavior for every model is specified somehow. Often this is only an informal specification in textual form that leaves space for interpretation and thus misunderstandings. The most precise way to specify the behavior of a system is giving clear invariant properties. They define the desired behavior and misbehaviors can be identified quickly.

There is a certain syntax introduced to express Linear Temporal Logic (LTL) formulas. Those formulas can be used to express this precise behavior for a model by setting properties for the system. Different quantors are defined for expressing constraints in the form of events that happen globally, eventually, until and in the next state of the execution. This allows to write properties that are supposed to hold for a system. Looking at a model for a traffic light system for pedestrians and cars, this might be for example the global property that cars and pedestrians never get a green light at the same time.



(a) Canonical form **(b)** Minimal form

Figure 3.5. Binary decision diagrams representing the formula $a \land (b \lor c)$

Faulty modeling causes these properties to be violated and the user is asked to change the model.

This kind of invariant checking could be implemented in KIELER, but this is out of the scope of this thesis. Different tools exist that fulfill this desired behavior for general state systems. The Symbolic Model Verifier (SMV) [1] is just one of many tools. It was one of the first model checking tools that uses binary decision diagrams. They were introduced by Bryant [Bry86] in order to represent boolean functions in a compact way. In Figure 3.5 two examples for binary decision diagrams are given. They both describe the formula $a \land (b \lor c)$ and in every level a variable is assigned to a truth value. In Figure 3.5(a) the outer right path sets a,b and c to true. The truth value of the formula is then also true. The outer left path sets all variables to false, thus the formula evaluates to false. Figure 3.5(b) is smaller. Setting a to false instantly causes the entire formula to be false and other variables are not set to a truth value.

Another approach is the use of boolean Satisfiability (SAT) solvers [MZ09]. They are programs that decide whether a formula can evaluate to true. This can be used to evaluate the LTL formulas in context with the model. This ensures that formulated specifications are fulfilled.

[1] urlhttp://www.cs.cmu.edu/ modelcheck/smv.html

Chapter 4

Used Technologies

The debugger is not an autonomous tool, and to understand the implementation details further technologies need to be explained more precisely. Functions are used that are implemented through the execution manager introduced by Motika [Mot09] called KIEM. The functionalities are illustrated below and its main components are also introduced.

Additionally, the debugger is embedded into the Eclipse Platform and thus has the possibility to use already implemented functions that are provided by Eclipse. The general structure of Eclipse applications is presented and especially debug features are pointed out.

4.1 KIEM

KIEM is important for the functions contributed by the debugger. A simulation and execution framework was added to KIELER by Motika [Mot09] so designed models can be tested in a theoretical environment with inputs provided manually.

Step, pause and run functionalities are already provided. In Figure 4.1 these buttons are shown. A step command performs one tick for the given model and then pauses. The run command steps continuously through the model with a specified delay in between the steps. The delay is entered in the field marked as cycle time in Figure 4.1. Pressing pause causes the run mode to pause. The stop button cancels the current execution and thus resets all current steps and visualizations. After pausing, the run mode can be restarted or the model can be stepped through by hand. In both modes the user can set inputs manually by loading a specific component to the execution manager prior to execution start. The concept of the components is described in the following.

Different components can be loaded that need to implement one of the two interfaces `IJSONObjectDataComponent` or `IJSONStringDataComponent`. Additionally they need to be registered as a data component through the extension point

`de.cau.cs.kieler.sim.kiem.json.datacomponent` (for JSONObjects) or

`de.cau.cs.kieler.sim.kiem.string.datacomponent` (for JSONStrings).

Those components can be added and removed and each provides its own functionality. Therefore the execution can be adjusted as needed. To understand the model a visualization of current active execution elements could be reasonable. For tests on execution times

a component that tracks the execution time and benchmarks it would be appropriate. Consequently, the execution can be extended generically by a lot of already implemented features.

In Figure 4.1 some of the components are illustrated but right clicking in the window enables choosing different components from a list. Some may visualize outputs or give the possibility to provide inputs. KIEM itself bundles them and organizes their execution. The component at the very top is the first one to get executed following the one below so execution is organized in a linear order. At some point there is a simulation component that handles the actual simulation. In Figure 4.1 this component is called SCCharts/SCG Simulator (C).

Data components in KIEM are defined as producer, observer or both. In Figure 4.1 a screen shot of the user interface is shown. This categorization is also specified next to the name of each component in the column Type. The data produced by a producer component is required by the components following the producer in the above mentioned linear order or in the following steps. In contrast, the observer does not influence subsequent components or steps, but reacts to the current state of execution. In case the cycle length is not long enough for observers to get executed, they may be skipped. Producers are not allowed to be skipped. In every step it is ensured that they will be called.

The debugger is implemented as a data component that is producer and observer. Hence, it can benefit from information given by other components. Especially a simulation component is required because it adds information of the current execution status and that is needed to determine potential pauses. Important components to understand the execution behavior of KIEM are presented and further explained below. Some are simply important for the simulation to be visualized or interactive, but the simulation component in particular plays an important role in connection with the debugger.

4.1.1 Data Table

There are two different kinds of Data Table components, but both visualize their behavior in the same data table window. In Figure 4.1 this table is marked, too. On the one hand inputs and outputs are presented and their emit status is marked. On the other hand the user has the opportunity to emit signals by hand, thus emulating the environment for the model.

As a consequence, those two different elements of the data table need to be placed chronologically around the simulation component explained in 4.1.3, so inputs can be provided before simulation and output is represented after execution. The component that enables setting signals manually is called `DataProducer` and therefore produces information so the simulation can read those. This component needs to be executed before the simulation. The component `DataObserver` observes the output signals that are

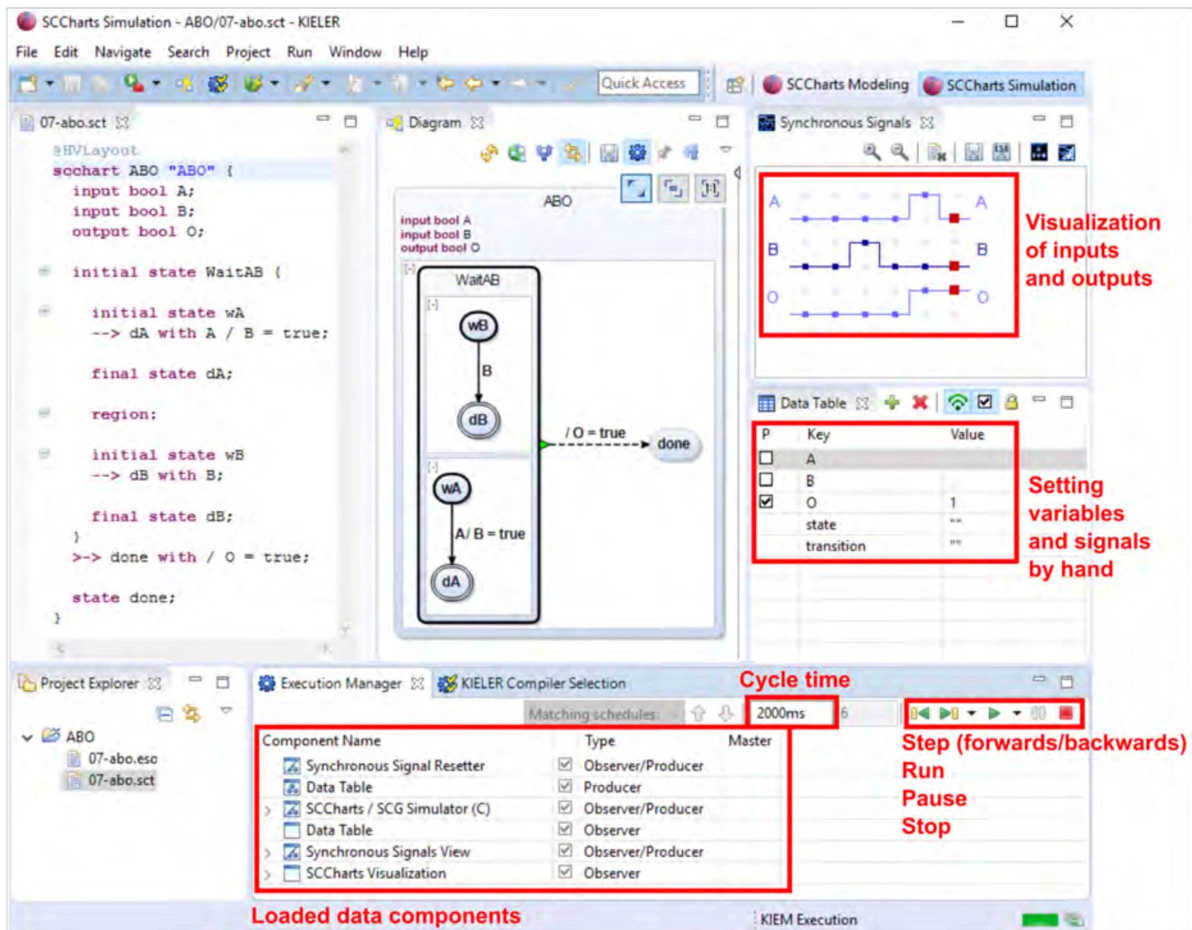


Figure 4.1. Overview of the elements that belong to KIEM highlighted in red rectangles

added by the simulation component. The output variables in the data table are thus set accordingly.

4.1.2 KART

KIELER Automated Regression Testing (KART) provides the opportunity to save execution traces as an .eso-file. The inputs and outputs at the specific tick instance are recorded and as consequence can be reused as input sequence for a following simulation or for defining tests for models. The former possibility is leveraged in this thesis. This component must be executed before the simulation component is activated.

4.1.3 Simulator

At the moment this component is entered, the actual simulation happens. There are different simulations predefined that can be used at this point like `C` or `Java`. The root SCCharts model is then transformed to the according code language and the generated code is compiled. This serves as the simulation background. All of the predefined simulations have in common that they receive inputs and provide outputs. The outputs of this simulation are saved and passed on to following components.

4.1.4 Synchronous Signal View

After one step of the simulation component the outputs are computed and so at this point they can be visualized. This component adds a window to KIELER labeled with Synchronous Signals. In Figure 4.1 in the top right corner this view is shown and the signal trace during this example execution is visualized.

4.2 Eclipse SDK

KIELER is integrated into the rich client platform Eclipse[1]. Eclipse SDK consists of the Eclipse platform, the Java Development Tooling (JDT) and the Plugin Developer Environment (PDE). It is designed for building integrated web and application development tooling. The platform itself does not provide many features itself but it is based on a plug-in model so that other features can be integrated quickly. Plug-ins are bundles of code or data that provide some functionalities to the overall system. Figure 4.2 gives a visual overview of the different subsystems in Eclipse SDK. The relevant parts are further explained in the following.

4.2.1 Eclipse Platform

The Eclipse Platform consists of many different subsystems. Those can be implemented in one or more plug-ins and are based on a runtime engine. The plug-ins that define a subsystem also provide extension points that can be used to add new behavior to the platform. The different components in the Eclipse Platform are described in the following whereas special attention is laid on the debug component.

[1] https://eclipse.org/

[2] http://help.eclipse.org/

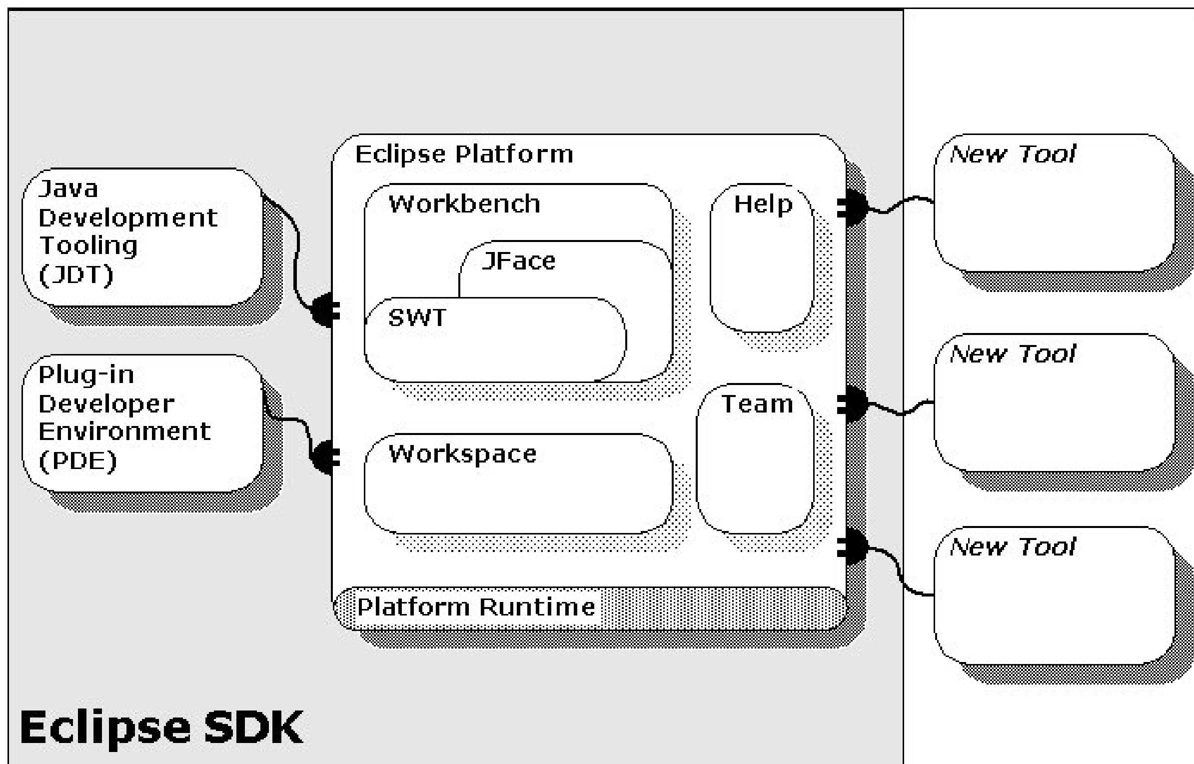


Figure 4.2. Overview of the Eclipse Platform Architecture[2]

Runtime

The runtime engine is implemented here. It is used to dynamically determine needed plug-ins and organize their execution. The main goal is to optimize the performance so no resources are wasted due to installed but not needed plug-ins.

Workspace

The resource plug-in is the main part of this element. It offers services for accessing folders, files etc. that the user wants to work on. Also plug-ins itself can create or modify projects, folders or even files.

Workbench

The main workbench interface is implemented here. The toolkits Standard Widget Toolkit (SWT) and JFace are used for user interface building and a lot of extension points are defined. This facilitates contributing elements to the menu, tool-bar actions, dialogs, wizards and many more.

Help

The help plug-in contributes a help web server and offers extension points so other plug-in documentation can be viewed as browsable book.

Team

The team plug-ins enable easy integration of features that facilitate team programming, repository access or versioning.

Debug

This element of the Eclipse Platform is the most interesting for this thesis because general debugging features are already accessible in this part of the platform. Other plug-ins are allowed to implement language-specific program launchers and debuggers. The launcher is not explained further because KIELER already provides a launcher with KIEM. The core debug plug-ins provide support for a generic debug model, debug events and listeners, breakpoint manager and an expression manager. These can be used and extended to be language-specific.

4.2.2 Plugin Developer Environment

The Plugin Developer Environment (PDE) is an extension of the Eclipse Platform. It provides tools to work with Eclipse plug-ins, fragments, features and update sites. It can be broken up into three components: UI, API Tools and Build.

The PDE UI provides editors, launchers, views and other tools that help improve the environment for creating new Eclipse plug-ins or similar features. One of the most used tools is the Form-Based Manifest Editor. It allows to edit all manifest files of a plug-in in a cleanly structured editor. Other features are for example the New Project Creation Wizard or Launchers that enable testing and debugging of Eclipse applications.

The API Tools provide help in documentation and maintenance of APIs provided by plug-ins. They include a Compatibility Analysis, Version Number Validation or the Javadoc `@since` tag.

Finally the PDE Build is used to create the automation of the plug-in build process. Ant scripts provide information on developement-time and manage the access of needed data by fetching it from a repository for example.

Chapter 5

The SCCharts Debugger

In KIELER the possibility to simulate a model is already given through KIEM and its components. However, for some purposes, especially for large models, debugging might still become a tedious task. The main focus is often on a special model element and it may be elaborate to manually run the simulation up to the point of its execution. A designed model might still include errors and every time the desired model element needs to be checked, other model elements need to be passed through.

The main part of this thesis is the implementation of a debugger in the context of KIELER.The underlying concepts of the new features are now presented. Figure 5.1 gives an overview of newly introduced visual components of the debugger. Firstly, it is discussed why KIEM is used and which advantages and disadvantages this brings.

5.1 Simulation Framework

In Section 4.1, the KIEM simulation framework is introduced. It provides the possibility to simulate a model and to extend the simulation with suitable components. I decided to use this simulation framework as the background for my debugger. Many useful features are already provided. In the following, the advantages and disadvantages are elucidated.

The simulation offers two different modes for execution. The model can be stepped through tick by tick and also a run mode is provided that does not wait for user interaction to continue execution. Both modes are useful for the debugger. Before reaching a breakpoint, the run mode is very convenient for fast forwarding through the model. After reaching the breakpoint, the user most likely wants the possibility to go through the model step by step. Additionally, taking a step back and reviewing the inputs and outputs is possible.

Many different components for this simulation framework already exist. Especially the KART component plays an important role in combination with the debugger. The possibility to read inputs from an .eso-file is already implemented and can be used without the need of further adjustments. Also, traces that are not saved in an .eso-file yet can be exported so a new .eso-file is created or the existing one is extended by one trace. In case this component would not exist already, a fast forward mode could not be realized without having to create further features that simulate an environment. Also

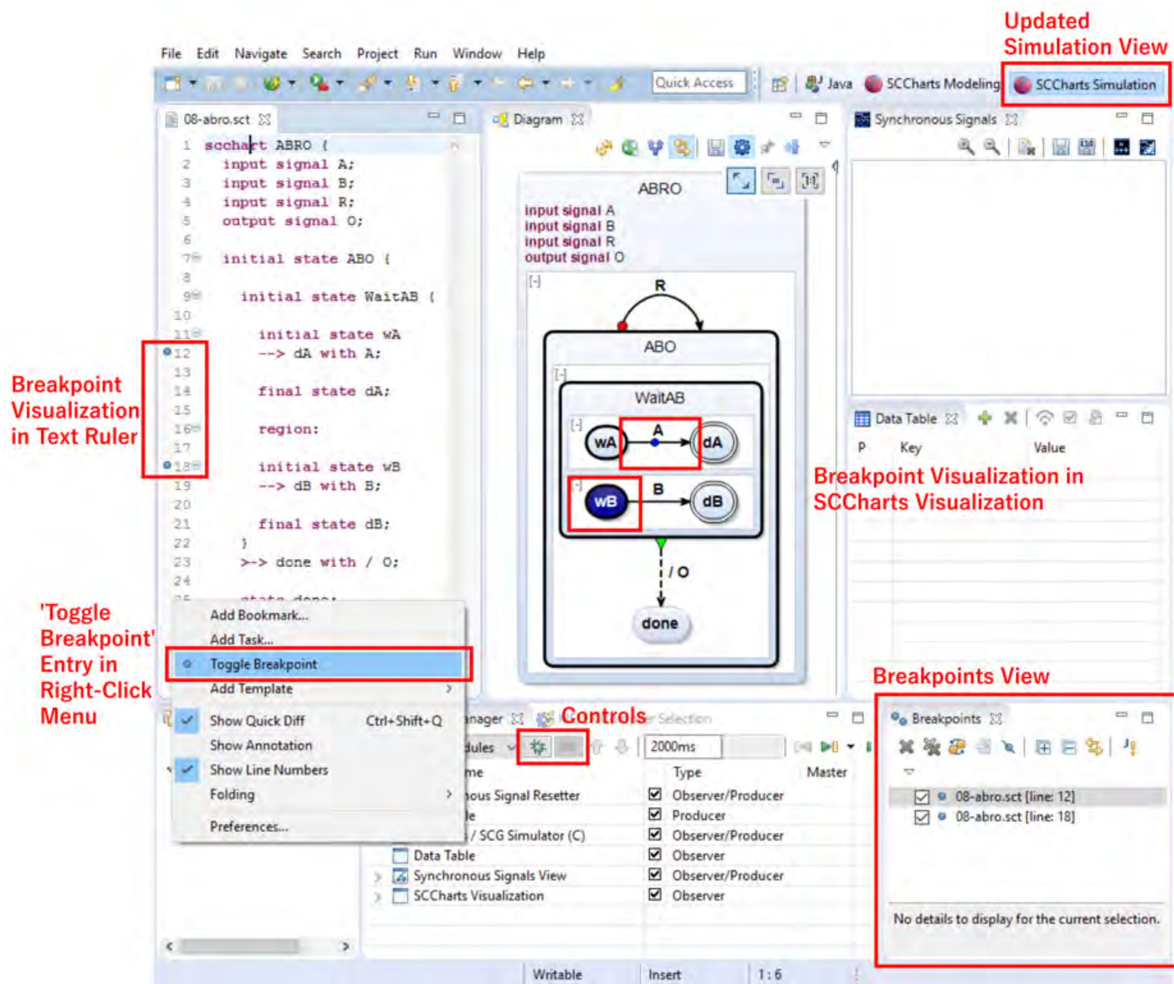


Figure 5.1. Screenshot of KIELER with the debugger elements highlighted with red rectangles

users are already familiar with the work flow of KIEM components and .eso-files. For many examples and tests especially .eso-files with many traces already exist.

However there are also disadvantages regarding the use of KIEM. The entire simulation is done through components in KIEM. Therefore the KIEM simulation component defines the granularity of the steps. It is not possible to make smaller steps in combination with immediate transitions at the moment. Hence only at the end of a step defined in the sense of KIEM the simulation can be paused.

5.2 Elements

An important feature for a debugger is easy handling and a self-explanatory meaning of icons. The next six subsections further explain the visual components and the reason why they are added.

5.2.1 Text Ruler

The main aspect thinking about breakpoints is the ability to place breakpoints somewhere in the code. A visual feedback is needed that gives further information about where in the given program code the breakpoint is set. In general Eclipse applications this is enabled due to double-clicking at the text ruler or choosing the entry in the right-click menu. It follows that a blue dot appears in the ruler at the mouse cursor's position.

In Figure 5.2 the modeling perspective in KIELER is shown. On the left side there is a textual editor that enables model editing in the SCCharts textual language and synchronizes a diagram view with it. The diagram view is shown on the right and it synthesizes the visualization of the SCCharts that are defined in the textual description.

The functionality of setting breakpoints in the editor is implemented through the debugger in KIELER. In the editor for the textual representation for SCCharts, breakpoints can be set. Breakpoints provide important information for the debugger component and only certain lines in the code are qualified to serve such information. Thus, only in those lines the possibility to specify breakpoints is provided. This can either be the textual description of a state or a transition. These are considered to be reasonable model elements. The current execution state of SCCharts is a state in the model and transitions manage the leaving and entering of new states. They are the core elements of state based modeling. The underlying semantic is further defined in Subsection 5.3.1.

5.2.2 Right-Click Menu

The functionality mentioned above is associated with a corresponding menu entry in the ruler right-click menu, thus it sets a breakpoint the line specified by the mouse cursor's position. Figure 5.1 shows the entry in the ruler right-click menu after Add Task....

5.2.3 Breakpoints View

In Eclipse applications, the debug perspective is also associated with the breakpoints view. This view is already implemented in the Eclipse Platform Debug UI plug-in and the breakpoints that are inserted in different resources are listed here. Also the user has the

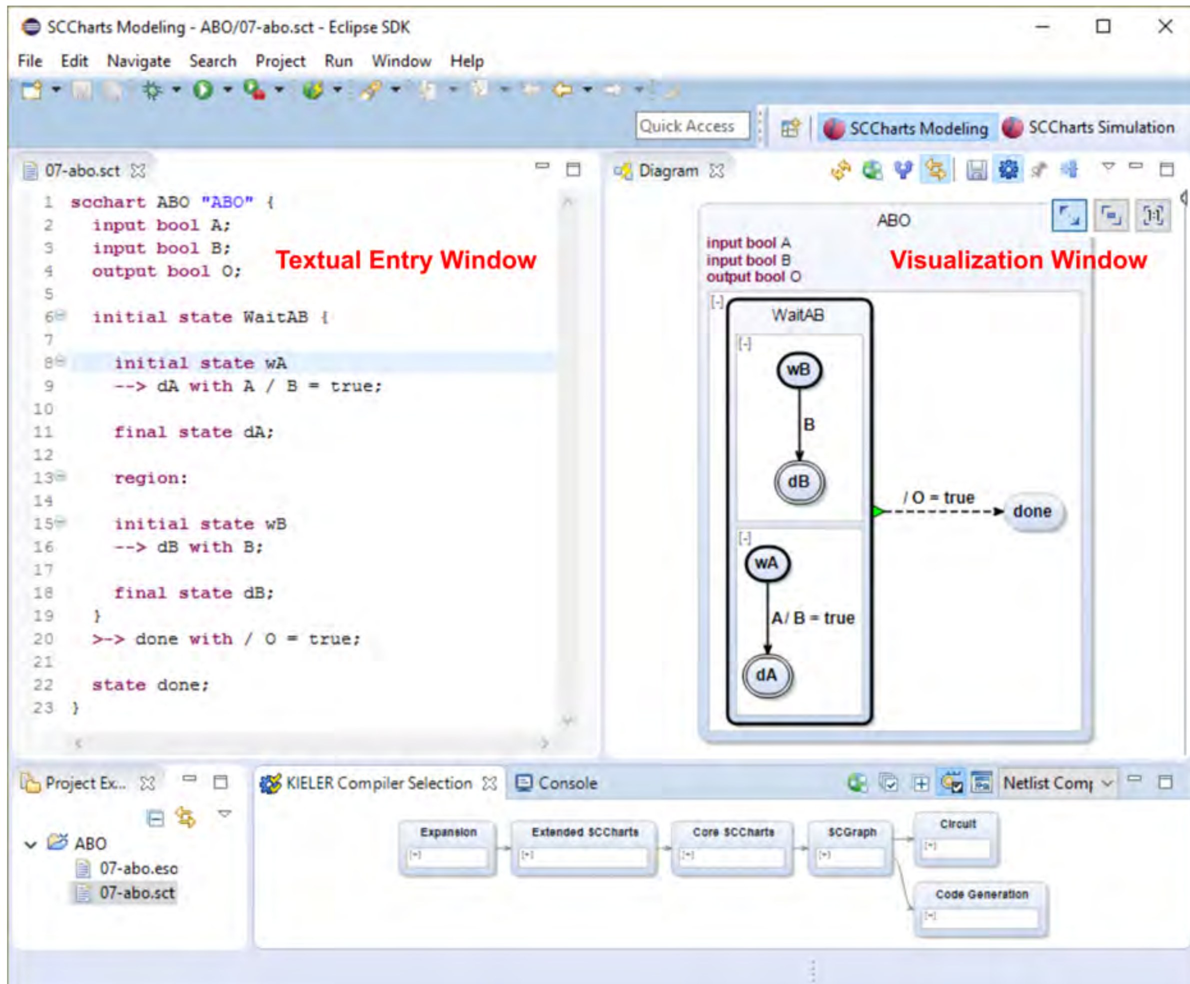
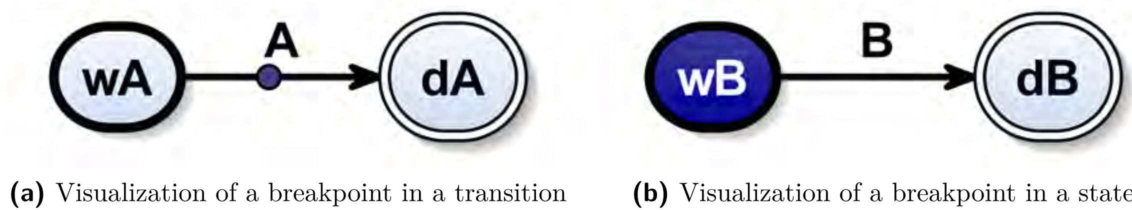


Figure 5.2. KIELER Modeling Perspective

option to disable or enable a breakpoint. Thus a disabled breakpoint is not considered as a reference to pause execution and is skipped.

5.2.4 Breakpoint Visualization

The textual syntax goes along with a visual representation as already mentioned in Section 1.1. The debugger now enables visualization of the breakpoint not only in the ruler context menu but also in the visualization. Thus, adding a breakpoint in the ruler context menu puts a dot in the line specified there and additionally the corresponding model element is visualized in the automatically layouted graph, the visual representation of SCCharts. This way the information added in the ruler context menu is processed to the visualization of the defined SCCharts in the diagram view. In Figure 5.3 the visualization



(a) Visualization of a breakpoint in a transition **(b)** Visualization of a breakpoint in a state

Figure 5.3. Visualization of breakpoints in SCCharts

of the two types of breakpoints is shown. A transition breakpoint adds a breakpoint-like circle on the transition arrow and a state breakpoint colors the entire state in dark blue and change the label color to white.

5.2.5 Controls

The Execution Manager KIEM already includes a control pane with buttons for running, stepping, pausing and stopping during simulation. In addition, the cycle time during run mode in the KIEM simulation can be set here. These controls are also shown in Figure 5.1 right next to the new debugger controls. For debugging reasons this pane is extended by two buttons. These two buttons are also used to control the simulation due to debugging functionalities and are illustrated in Figure 5.1.

This button can be toggled. When enabled, the debug mode is turned on, meaning that during simulation the active model element is checked for a breakpoint. If it contains a breakpoint, the simulation in KIEM is paused after the end of the current active step.

This button can only be toggled when the debug mode is turned on, otherwise it is disabled. Whether it is activated or not indicates if the execution is fast forwarded or not. Fast forwarding means setting the cycle time of one step to zero so no unnecessary delay is produced when simulating the model.

5.2.6 Simulation Perspective

The KIEM control window is already included in the SCCharts simulation perspective. For debugging purposes, the breakpoints view mentioned in 5.2.3 is now included in this perspective, too. As a consequence, when resetting the perspective the view is placed at the bottom right as Figure 5.1 shows.

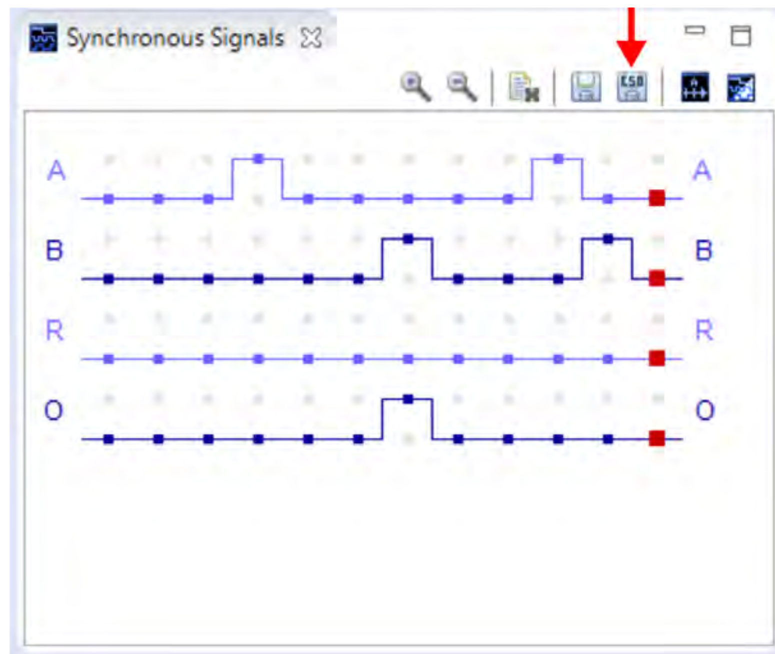


Figure 5.4. Synchronous Signals View showing the button used to save an .eso-file

5.3 Usage

In the following, it is further explained how the general debugging work flow is presented and how the new features can be used. Newly introduced elements are presented and further explained. Implementation details are given in Chapter 6.

5.3.1 Adding a breakpoint

In order to debug something using the KIELER debugger, a breakpoint must be added to the model. A breakpoint is only allowed in a line that contains a reasonable context for a breakpoint, like a state or a transition. Other lines do not serve reasonable information for the simulation to be paused. The user can double click at the line in the ruler menu or right-click and then chose the Toggle Breakpoint entry as shown in Figure 5.1 to set a breakpoint. The definition of a state and a transition breakpoint is given below. Afterwards, a blue dot is shown in that line indicating that there is a breakpoint and also the breakpoint is added to the breakpoint view as a new entry. In the following it is explained how breakpoints at a state and at a transition are defined.

State Breakpoint

State Breakpoints are placed in the line defining a state. In Figure 5.5 an example for an SCCharts model is shown that illustrates the definition of a state breakpoint. In Figure 5.5(a) the visual representation is shown and in Figure 5.5(b) the corresponding textual description is presented. In line 11 containing the text `state C` a breakpoint is specified.

The model is executed and in the first three ticks an output is set to true. Since we do not receive any inputs, this behavior is always the same. After the first tick output `O1` is set, after the second tick output `O2` is also set to true, so after tick two (see tick three) both `O1` and `O2` are set to true. After the third tick finally output `O3` is also set so all three outputs are true.

When this model is simulated in run mode with debug mode turned on, the execution is paused after entering state `C`. Figure 5.5(c) visualizes the execution trace of the moment the model is paused. Output `O2` was set but `O3` was not yet set, which is the expected outcome for stopping the simulation when entering state `C`.

Another interesting case is a breakpoint in a hierarchical state. In Figure 5.6(a) the state `C` is hierarchical with two regions in it. On the state `C11` there is a breakpoint. When in region `C1` the transition is taken, then output `CO1` is set to true. In region `C2` output `CO2` is set when the transition in this region is taken.

Simulating this model causes the execution to stop as soon as `C11` is entered. In Figure 5.6(b) the execution trace at the moment the simulation pauses is shown. Neither output `CO1` nor `CO2` is set to true, so no transition inside the regions was taken yet. Hence the simulation was paused because of entering state `C11`.

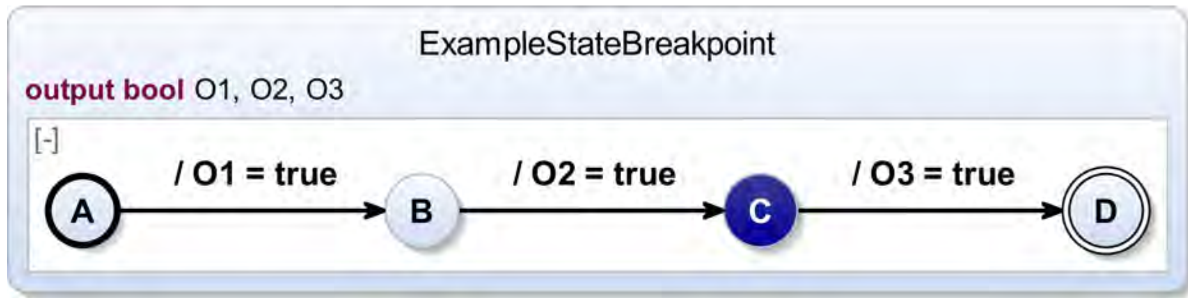
In conclusion, on entering hierarchical states the state itself is first checked. If it does not contain a breakpoint, all its initial states are checked for a breakpoint recursively until a breakpoint is found or non-hierarchical states are reached.

Transition Breakpoint

A transition breakpoint can be placed at any kind of transition. In Figure 5.7(a) an example for a SCCharts model is shown that illustrates the definition of a transition breakpoint.

The SCCharts model is the same as explained in Section 5.3.1. The only difference is the breakpoint itself. It is now specified at the transition `/ O2 = true` going from `B` to `C`. In Figure 5.7(b) the corresponding editor is shown and in line 9 containing the text`--> C with /O2 = true` a breakpoint is set.

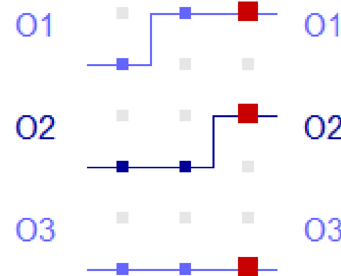
The simulation of this model also has the same output trace as in Figure 5.5(c) illustrated for the state breakpoint. The difference is the reason for the pausing. After leaving state B the breakpoint transition with `/ O2 = true` is taken. At this moment it



(a) SCCharts model with a state breakpoint

```
1  scchart ExampleStateBreakpoint {
2
3    output bool O1, O2, O3;
4
5    initial state A
6    --> B with /O1 = true;
7
8    state B
9    --> C with /O2 = true;
10
11   state C
12   --> D with /O3 = true;
13
14   final state D;
15 }
```

(b) Textual Editor of this model with a state containing a breakpoint



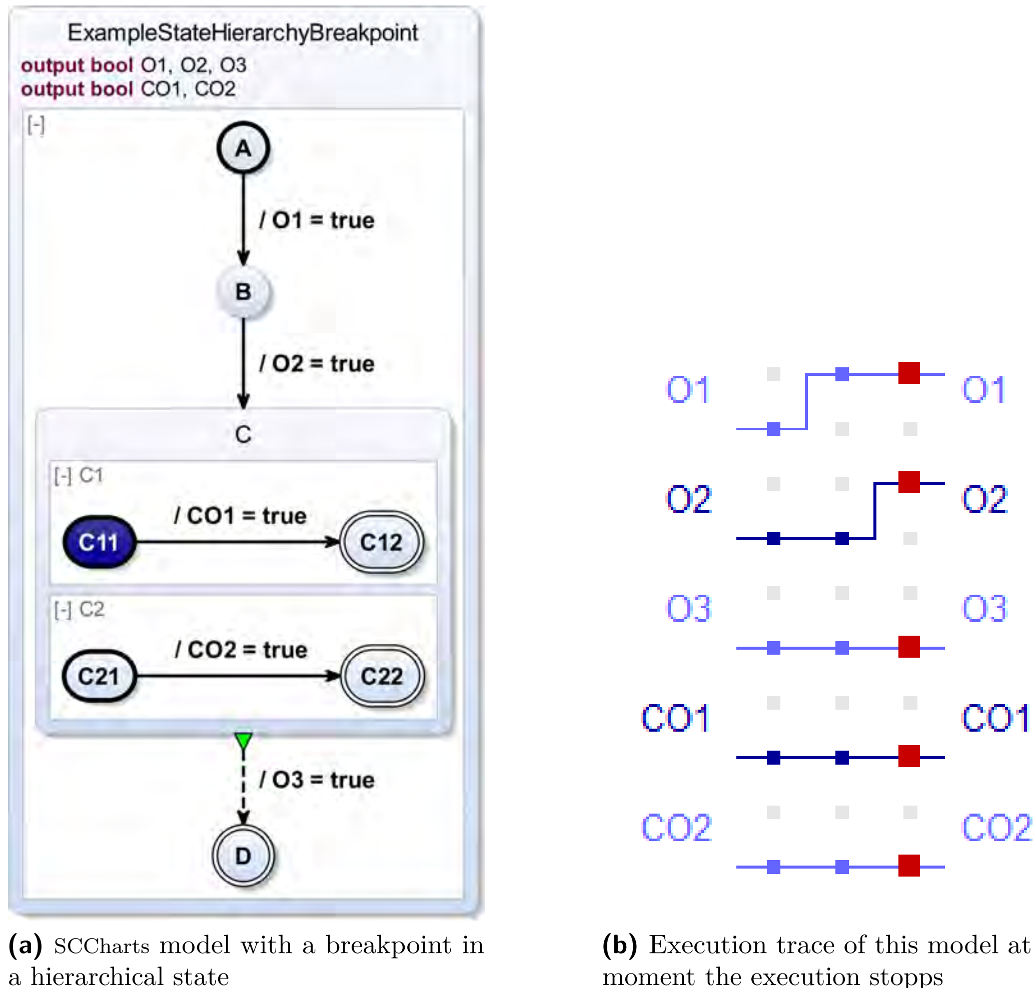
(c) Simulation trace of the model at the moment it pauses due to the breakpoint

Figure 5.5. Semantics of a state breakpoint

is realized that the simulation must be paused and at the end of this step. This is the reason for the pausing.

Moreover the execution is paused after a transition is taken and not prior to it. Pausing the execution prior to it would enable the modification of inputs and thus it would be possible that the model reacts differently to these newly provided inputs and consequently would not run into the breakpoint. Therefore the reason the execution was paused initially was removed and with the new inputs the pause would even be incorrect.

Note that KIEM defines the moments when pausing is possible and not the debugger itself. Adjustments to KIEM that enable a different definition of a step would also cause the debugger to operate on this definition. Additionally, other positions for breakpoint might be reasonable like regions. However, this does not provide new pause moments for the debugger, because a breakpoint on a region can be equivalently created by a breakpoint on the initial state in this region.



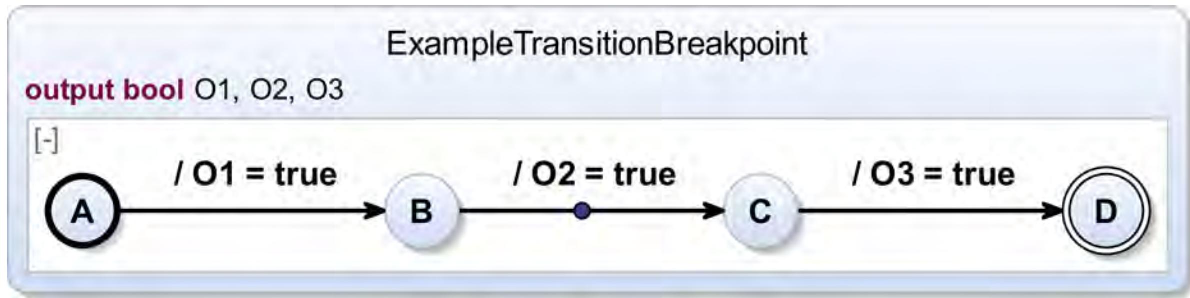
(a) SCCharts model with a breakpoint in a hierarchical state

(b) Execution trace of this model at the moment the execution stopps

Figure 5.6. Semantics of a state breakpoint with hierarchical states

5.3.2 Creating an .eso-file

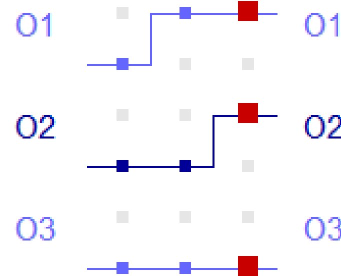
For a model to be able to get executed without user interaction there must be a possibility to provide input information. This functionality is already implemented and can be provided through .eso-files. Thus, an .eso-file is needed to use the debugger because the debugger includes the KART component. It is reasonable to require this file because fast forwarding would not be possible without it. If the file is missing the user is notified through KART that the file is missing but needed. In case the user does not need any inputs, it is recommended to delete the KART data component in the execution manager. This way no .eso-file is required.



(a) SCCharts model with a transition breakpoint

```
1  scchart ExampleTransitionBreakpoint {
2
3    output bool O1, O2, O3;
4
5    initial state A
6    --> B with /O1 = true;
7
8    state B
9    --> C with /O2 = true;
10
11   state C
12   --> D with /O3 = true;
13
14   final state D;
15 }
```

(b) Textual Editor of this model with a transition containing a breakpoint



(c) Simulation trace of the model at the moment it pauses due to the breakpoint

Figure 5.7. Semantics of a transition breakpoint

The .eso-files in general contain traces for the execution and this way it can be specified at which tick signals are emitted. In Figure 5.8 an example for an .eso-file is shown. The reset command separates different traces.

In the KART data component it can be configured which .eso-file is supposed to be loaded for execution and additionally which trace should be loaded, while zero meaning the first trace. Figure 5.9 shows this menu.

To create such a file the simulation must be run manually one time. The desired trace is created by setting inputs by hand through the data component. When the desired model element is reached, the trace can be saved. To do so, there is a save button in the Synchronous Signals view in the upper control pane and the needed input signals can be selected. In Figure 5.4 this button is shown. The .eso-file should then be saved in the same project where the model file is located. If an existing .eso-file is overwritten, the new trace is appended to the end of the old one.

```
1  ! reset;
2
3  % Output:
4  ;
5  B
6  % Output:
7  ;
8  A
9  % Output: 0
10 ;
11
12 % Output:
13 ;
14 ! reset;
15 A
16 % Output:
17 ;
18 A
19 % Output: 0
20 ;
21 A
22 % Output:
23 ;
24 A
25 % Output:
26 ;
27 ! reset;
28
29 % Output:
30 ;
31
32 % Output:
33 ;
34
35 % Output:
36 ;
37 B
38 % Output:
39 ;
40
41 % Output:
42 ;
43
44 % Output:
45 ;
46 A
47 % Output: 0
48 ;
49
50 % Output:
51 ;
52
53 % Output:
54 ;
55 ! reset;
56 A
57 % Output:
58 ;
59 A
60 % Output: 0
61 ;
62 A
63 % Output:
64 ;
65 A
66 % Output:
67 ;
```

Figure 5.8. Example of an .eso file with four different traces

Component Name / Key	Value		Type	Master
KART - Replay/Record Input		☑	Observer/Producer	
ESO Model File	[ACTIVE EDITOR]			
Trace number to replay	0			
Training mode	false			
Configuration variable	kartConfig			
Output signals/variables variable na	kartOutput			
Previous input signals variable nam	kartPrevInput			
Automatic stop and increment tace	true			
SCCharts / SCG Simulator (C)		☑	Observer/Producer	

Figure 5.9. KART configuration menu showing where the .eso-trace number is listed

5.3.3 Schedules

Different schedules can be chosen in the KIELER Execution Manager (KIEM). An example for the menu that pops up is shown in Figure 5.11. A schedule includes the corresponding data components and their configuration such as the cycle time for a tick in run mode.

This chosen schedule can now be extended using the newly added debug button. There are two data components that are added to the current schedule when enabling the debug mode. The first is the general debug component that is further explained in the implementation Chapter 6. The second one is a component named KART that takes input information from the created .eso-file. If no .eso-file is provided, a dialog pops up as shown in Figure 5.10.

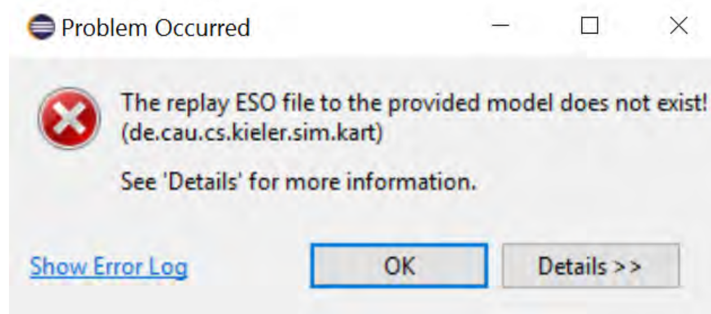


Figure 5.10. Pop-up dialog shown when no .eso-file is provided

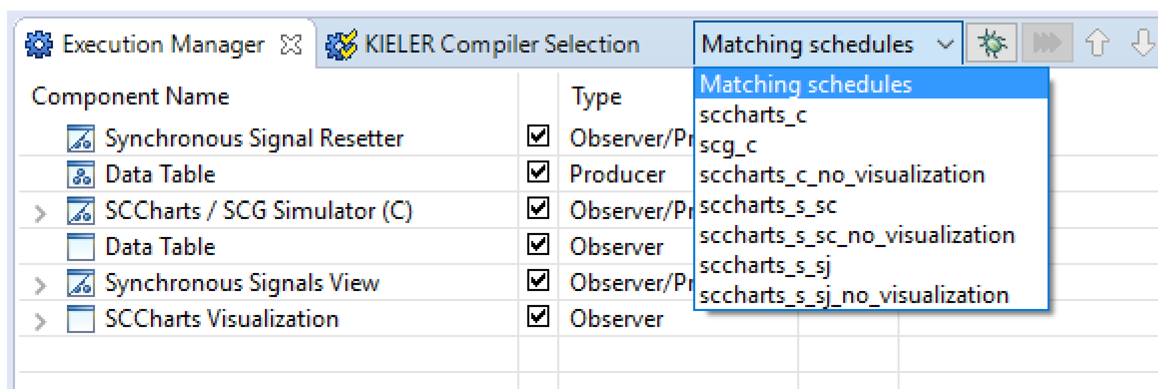


Figure 5.11. Example of a schedule menu

5.3.4 Starting Execution

Finally, the execution can be started. There are two different possibilities to do so and both only make sense in KIEM run mode, since in step mode the execution is paused after each step anyway. After starting the simulation, the choice of data components is fixed and can only be changed after stopping the execution. Hence the user has to decide to turn on the debug mode prior to starting the simulation, and after starting, the button is disabled. The fast forward mode can be turned on or off at an arbitrary time as long as the debug mode is on.

If the fast forward mode is turned off, the execution is run with the delay specified in the cycle time field. The user does not need to manually step through the execution but also it is not forwarded in time. Either way the execution is paused when a breakpoint trace is reached.

The second option is the fast forward mode turned on. This way everything is skipped with as little delay as possible, and just when the execution reaches a point where a breakpoint is specified, it is paused.

In both options it is important to mention that the .eso-files are not of unlimited length. In case the end is reached, simply no inputs are provided any longer. The model reacts the way it would without inputs set and thus the model possibly runs in an infinite loop if there also is no reason to pause due to a breakpoint.

Chapter 6

Implementation

In this chapter implementation details for the debugger in KIELER are given. It is explained which technologies are used and how they are embedded for the use of a debugger. The presentation of the implementation is led by four main goals which are defined in the following, and subsequently it is explained what is implemented to fulfill these goals.

The debugger for the KIELER KIEM was implemented in the plug-in `de.cau.cs.kieler.sccharts.debug`. This plug-in has two different main packages with which it is split into visual parts and non-visual parts. Also some functionalities do not require further implementations but are realized through extension points. In the following the desired debugger features are connected with the way they are implemented.

6.1 Implementation Goals

In Section 1.3 the intentions and requirements for the debugger are stated. In the following the process of the debugger development is illustrated and implementation details are given. The major user actions provided by the debugger are listed below. In the presentation of implementation details it is stated at which point one of these goals is reached.

(1) Define a breakpoint in the editor.

(2) Visualize the breakpoint in visualization of SCCharts.

(3) Simulate the model and pause at breakpoint position.

(4) Extend the simulation perspective with breakpoints view.

Figure 6.1 gives an overview of the used extension points and relates them to functional aspects of the implementation. Also contributions to the user interface are pointed out by *+UI*.

org.eclipse.ui.menus
org.eclipse.ui.commands
org.eclipse.ui.editorActions
+ UI
Ruler Right-Click Entry

org.eclipse.debug.core.breakpoints
org.eclipse.core.resource.markers
org.eclipse.core.runtime.adapters
org.eclipse.debug.ui.debugModelPresentation
+ UI
Breakpoint Model

de.cau.cs.kieler.sim.kiem.json.datacomponent
de.cau.cs.kieler.sim.kiem.config.DefaultScheduleContributor
de.cau.cs.kieler.sim.kiem.toolbarContributor
de.cau.cs.kieler.sim.kiem.eventListener
+ UI
Register with KIEM and UI

org.eclipse.ui.perspectiveExtension
+ UI
Extended Simulation Perspective

de.cau.cs.kieler.sccharts.klighd.hooks
+ UI
Hook for SCCharts Synthesis

Figure 6.1. Extension Points used in `de.cau.cs.kieler.sccharts.debug` grouped by their function

6.2 Preliminaries in Editor

The first step towards a debugger is to provide a way to define breakpoints. For all newly introduced elements of this thesis, the goal was to provide an Eclipse-like look and feel. Hence the setting of breakpoints is realized through the Eclipse ruler menu.

This requires a definition of a breakpoint first. This is done by the class `SCCharts-Breakpoint` which identifies all breakpoints as `LineBreakpoint`. Also the extension point `org.eclipse.debug.core.breakpoints` is used to register the breakpoint. In Figure 6.1 the group Breakpoint Model lists all used extension points for setting up an appropriate debug model.

This breakpoint now needs to be associated with the XText Editor for .sct-files so the ruler is able to represent breakpoints. For this purpose, an adapter is implemented that implements `IToggleBreakpointsTarget` and provides information on creation and deletion of breakpoints through required methods. The class `SCChartsBreakpointTargetAdapter`

```
1 private void delegateToggleToTarget() {
2  IEditorPart editor = PlatformUI.getWorkbench().getActiveWorkbenchWindow()
3             .getActivePage().getActiveEditor();
4
5   SCChartsBreakpointTargetAdapterFactory factory =
6       new SCChartsBreakpointTargetAdapterFactory();
7
8   SCChartsBreakpointTargetAdapter target = (SCChartsBreakpointTargetAdapter) factory
9             .getAdapter(editor, SCChartsBreakpointTargetAdapter.class);
10
11   try {
12      target.toggleLineBreakpoints(null, null);
13      Display.getDefault().update();
14   } catch (CoreException e) {
15      e.printStackTrace();
16   }
17 }
```

Figure 6.2. Method `delegateToggleToTarget()` of class `TextRulerHandler`

implements this functionality. It defines only `LineBreakpoint` as a valid breakpoint (no method or watchpoint breakpoints) and the toggle event `toggleLineBreakpoints(...)` is also implemented which defines removing and creating of a line breakpoint.

Now breakpoints exist and the editor is able to represent them. However breakpoints cannot be specified yet. For this purpose the right-click entry in the ruler context menu Toggle Breakpoint is added. This is done through extension points. The ones used to realize this feature are shown in Figure 6.1. The group Ruler Right-Click Entry represents the extension points used for creating the connection to the editor. The last extension point in the list `org.eclipse.ui.editorActions` is the one that enables double-clicking on the rule as an optional way of specifying a breakpoint in a line. The class `TextRulerHandler` is the one that provides the methods that are executed on a toggle event.

Figure 6.2 illustrates the main functionality. First the active editor is accessed and an instance of a factory is initialized that creates an instance of the XText editor adapter `SCChartsBreakpointTargetAdapter`. This instance is realized in combination with the editor and the factory. The above mentioned method `toggleLineBreakpoints(...)` method is then invoked and the interface is updated. Furthermore in the `toggleLineBreakpoints(...)` it is also determined whether the specified line is qualified for a breakpoint. A qualified line for a breakpoint is one that either defines a state or a transition. At this point goal (1) is achieved. Breakpoints can be added and deleted in the editor and they have a specific definition for SCCharts breakpoints.

Table 6.1. Methods implemented in `BreakpointListener`

`breakpointAdded(...)`	In case the breakpoint is enabled, the corresponding `EObject` is resolved and `handleHighlight(...)` is invoked with indication that highlighting must be added.
`breakpointRemoved(...)`	The corresponding `EObject` is resolved and `handleHighlight(...)` is invoked with indication that highlighting must be removed.
`breakpointChanged(...)`	If something was changed it is checked whether the breakpoint is enabled or disabled now. The `handleHighlight(...)` in invoked according to the result so either the highlighting is added or removed.

6.3 Preliminaries in Visualization

A breakpoint is not only shown in the ruler but I also added a highlighting in the visualized SCCharts. Figure 5.3 illustrates how transition and state breakpoints are represented. States are highlighted by adding a new background style in dark blue and changing the label color to white. Transitions are visualized by adding a decorator that is a blue circle in the middle of the transition line.

The visualization components are placed in the class `BreakpointVisualizationHook`. This class extends the `SynthesisHook`, thus allowing to hook into the SCCharts synthesis. After performing the synthesis of the model, for all breakpoints a highlighting method is invoked. This method is called `handleHighlight(EObject, boolean)` and according to the object that is supposed to get highlighted, either a method for states or one for transitions is called. The boolean flag indicates whether the highlighting is added or removed.

At this point the visualization after a synthesis event is enabled but adding a breakpoint does not invoke a new synthesis of the model. Hence, adding a breakpoint does not show the corresponding visualization in the diagram view until the model is saved. The next focus is on enabling immediate visualization results.

In order to provide this functionality, a listener on breakpoint events is needed. This listener is realized through `BreakpointListener`. Note that disabling a breakpoint also causes the highlighting of the breakpoint in the visualization to be removed. In Table 6.1 the provided methods are listed and their effects are explained. Now also the goal (2) is realized. The preliminaries are fulfilled and in the following the focus is on the simulation.

6.4 Simulation

The simulation of models is already implemented through KIEM. As already mentioned in Section 1.1.2, data components can be added to the execution manager and they get executed in a linear order. In order to provide new features in the simulation, the implementation of a new data component that serves as a debugger is realized. This way it is seamlessly integrated into KIEM and the execution setup can also be saved in a .execution file for future reuse in case of modification.

For every data component a `step(...)` method must be implemented that provides the functionalities needed in every step of the execution. This method is where the model needs to be checked and eventually simulation must be paused. Note that the only possible positions to pause execution are after each step and KIEM defines the range of a step. Hence, breakpoints on immediate transitions cannot pause the execution at this exact moment but at the end of the performing tick.

The debugger is realized in the context of KIEM. The class `DataComponent` represents the KIEM component. It implements the interface `IJSONObjectDataComponent` and this interface requires the `step(JSONObject)` method. Also it extends `JSONObjectDataComponent`. This is needed as an implementation for the extension point of the KIEM data components. It also requires `DataComponent` to implement the methods `initialize()`, `wrapup()`, `isProducer()`, `isObserver()` and `getDataComponentId()`. In Table 6.2 the context of these methods is further explained and also their implementation is described. The `step(...)` method is explained in more detail in the following.

In Listing 6.3 an excerpt of the step method is shown. Firstly, the argument `jSONObject` needs to be evaluated. In case the model took a transition, this transition is added to this object with `"transition":` marking the beginning of the list of active transitions. These active transition are filtered into an array in line 10. In line 17 to 20 this string representation is translated to the actual `EObject`, the transition object itself. All active transition objects are added to a list (line 23) and passed on to the `checkOnPause(...)` method.

The `checkOnPause(...)` method now checks for all objects if there is a breakpoint associated with it. Note that all objects that are passed in the arguments list are transitions. For every transition the target state is fetched. For this state and all its hierarchical nested states that may be entered (initial states) it is then checked if they contain a breakpoint. In both cases if there was a breakpoint associated with an object, the execution is paused. Finally, the goal (3) is also achieved.

Table 6.2. Significant Methods in `DataComponent`

`initialize()`	This method is invoked prior to the `step(...)` method. The currently active model is fetched from the KIEM plug-in so the step method operates on the same model that KIEM does.
`wrapup()`	This methods defines what needs to be wrapped up after the execution is finished. There is nothing to do here.
`isProducer()`	Defines whether the component is a producer. The debugger is a producer and thus it is ensured that it is called in every step.
`isObserver()`	In case this is false, the `step(...)` method does not receive any arguments. The debugger needs to react to things that happen in the current step. Hence the JSONObject with information on the execution is needed and so the debugger also needs to be an observer.
`getDataComponentId()`	When the current configuration of the execution manager is saved in an .execution-file, this method is called for all data components. In the default implementation this would also encode whether this is a producer or observer. In case this changes for some reason, all existing .execution-files need to be updated. For this reason this method is overwritten so this is not included in the encoding for the .execution-file.
`step(JSONObject)`	This is the heart of the debugger. All debugger work is done or initiated here.

6.5 Perspective

At this point, breakpoints are completely implemented. There is only goal (4) left to reach which inserts the breakpoints view in the perspective of the simulation.

The simulation perspective is defined in the plug-in `de.cau.cs.kieler.core.perspectives`. The extension point `org.eclipse.ui.perspectives` is used and creates the perspective SC-Charts Simulation. Its identifier is set to `de.cau.cs.kieler.simulation`. This defined perspective can be extended using the extension point `org.eclipse.ui.perspectiveExtension`. This extension point requires the identifier of the perspective it is supposed to extend. Hence, the newly added view is identified and can be connected with the breakpoints view. This view is defined in `org.eclipse.debug.ui.BreakpointView`. The breakpoints view is now placed at the bottom right corner by default. Finally all goals are achieved and the implementation of the debugger is finished.

```
9  String transitionString = jSONObject.get(transitionKey).toString();
10 String[] transitions = transitionString.replaceAll("\\s", "").split(",");
11 
12 for (String transition : transitions) {
13     if (transition.length() > 1) {
14        if (resource == null) {
15          update();
16        }
17        EObject active = resource.getEObject(transition);
18        if (active == null) {
19           active = getEObject(transition);
20        }
21        if (active != null) {
22           if (!contains(statesByStep, active)) {
23              currentStepObjects.add(active);
24           }
25        }
26    }
27 }
28 if (!currentStepObjects.isEmpty()) {
29    checkOnPause(currentStepObjects);
30 }
```

Figure 6.3. Extract of the `step(...)` method that illustrates how the transition objects are pulled out of the JSON String and prepared for the `checkOnPause(...)` method.

Chapter 7

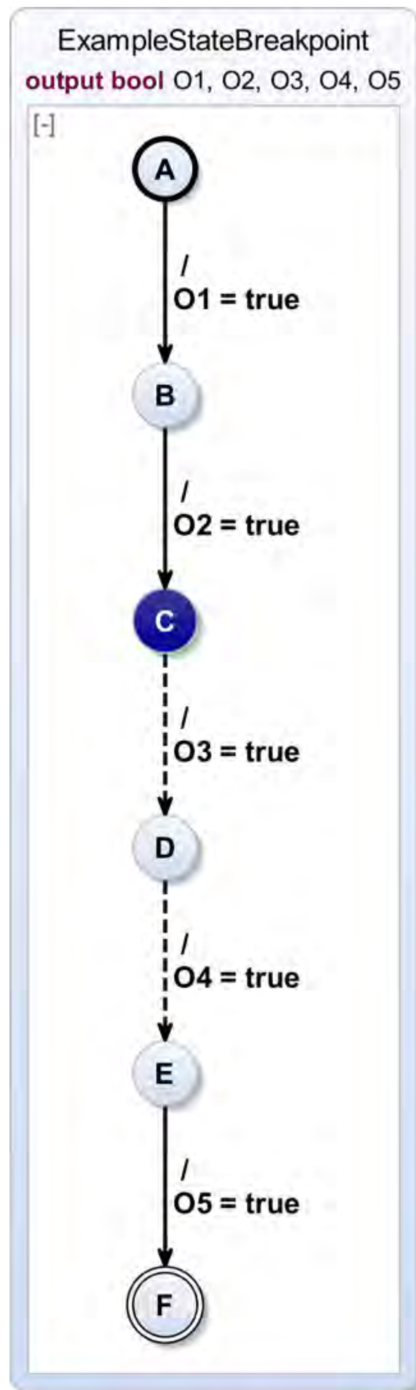
Evaluation

This chapter analyses the usability of the KIELER debugger. Section 7.1 discusses the limits of the debugger and circumstances where halting cannot be achieved at the correct model position. Section 7.2 introduces a real-world example of a bar code reader. Furthermore, the debugger is tested according to some potential problem situations. Some of the factors that are contemplated are simplicity in usage, error finding potential and the integration in KIELER. Note that the debugging is not performed while execution runs on the robot. The KIEM simulation is used that emulates the behavior of the robot, therefore there might still be influences in the real Mindstorm robot context that are not contemplated and cannot be foreseen.

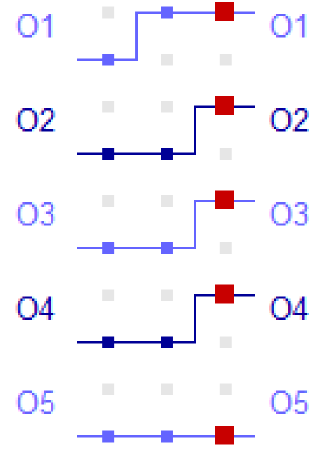
7.1 Limitations

In Section 5.1, the limitations of the KIEM framework and the advantages and disadvantages are discussed. These limitations directly affect the debugger. This is followed by the introduction of an example that demonstrates, how the limitations of KIEM can influence the debugger. Immediate transitions form a problem, because they are not stepped through one by one. At the end of a tick an arbitrary amount of immediate transitions may have been taken.

In Figure 7.1, an SCCharts model is shown that portrays this problem. The state C is defined as a breakpoint. However, the simulation cannot be paused at the exact moment this state is entered. The immediate transition leading from C to D and the one from D to E are taken before the simulation can be paused. The moment when entering state E is the first moment that the simulation can pause. The immediate transitions are taken in the same tick as the state C is entered and therefore this marks the end of this tick. It is important to think of this limitation when debugging SCCharts. Setting the breakpoint one step prior, at state B, allows to manually step and examine the immediate behavior.



(a) SCCharts model with a chain of two immediate transitions after a breakpoint



(b) Signals view at the moment the simulation is stopped

Figure 7.1. Problems of SCCharts breakpoints with immediate transitions

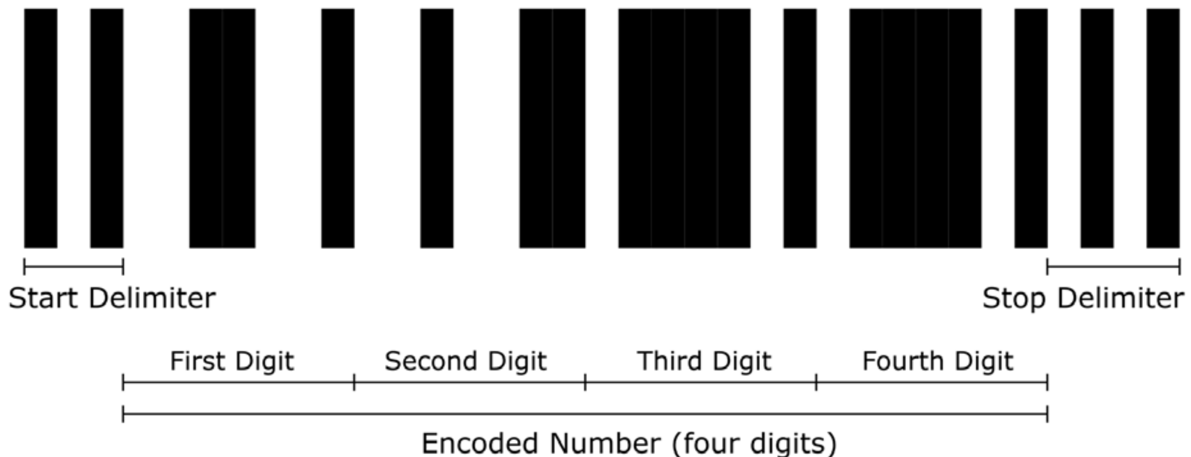


Figure 7.2. Structure of a bar code

7.2 Testing with a Bar Code Reader

In Appendix A, a model for a bar code reader is given. It is developed to run on an NXT LEGO Mindstorm Robot. This model is further explained in Section 7.2.2. Additionally, I go through potential problem situations and analyze their debugging process.

7.2.1 Prerequisites and Problem Statement

The robot is connected with different sensors. There are two motors attached and they handle the movement and the speed. Additionally, there is a touch sensor and a light sensor connected. The light sensor is aimed to the ground.

The bar code reader problem is stated as follows. At the beginning, the robot calibrates itself. He saves a representation for dark and a representation for light. Afterwards, it drives forward until a sequence of a dark line, a light line and a dark line occurs. This marks the start delimiter. After this delimiter is finished, four digits are read. Each of the digits is represented by seven lines. The sequence of dark and light lines encodes the digit and after four digits a stop delimiter follows. This is represented by a sequence of light, dark, light, and dark lines. Figure 7.2 summarizes the format of a bar code.

The decoded number read defines the turn angle whereas the four digits encode the number they represent. After reading the first bar code, the robot is supposed to turn according to the number read. A negative number corresponds to a counterclockwise turn and therefore a positive number represents a clockwise turn. The encoding of the number is defined as follows.

The first three digits $d1, d2, d3$ build a number y with $0 \leqslant y \leqslant 999$. The number x is computed by $x = y < 500?\ y : y - 1000$ and this is the actual translation of the number

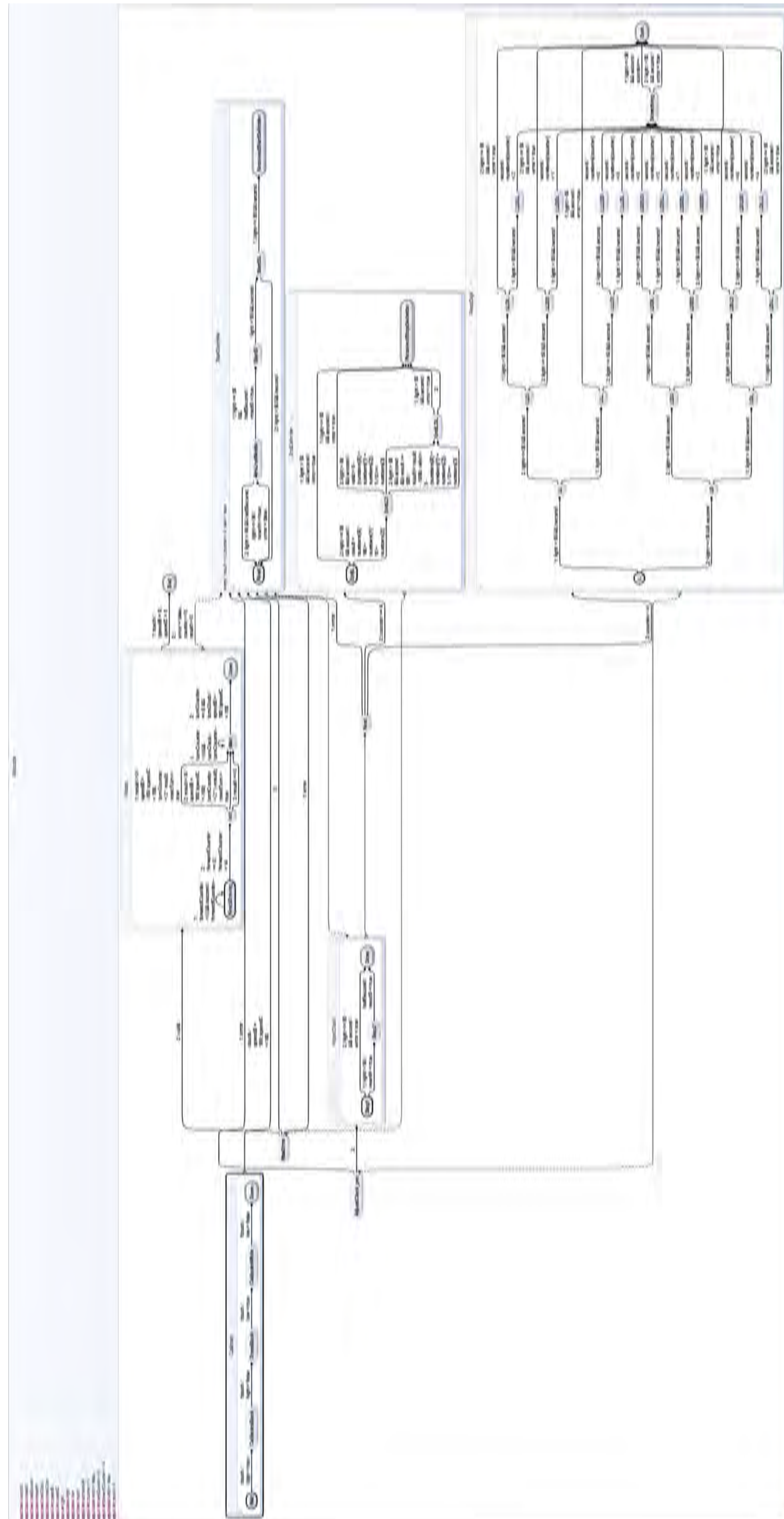


Figure 7.3. SCCharts model of the bar code reader

encoded. Therefore, the number range is $-500 \leqslant x \leqslant 499$. The last digit $d4$ of the bar code is a parity bit. A valid code satisfies $d4 = 9 - ((d1 + d2 + d3) \bmod 10)$. Consequently 1233 is a valid code, but 1234 is not. The encoding of the number 1233 can be seen in Figure 7.2. After the turn, the robot starts driving forward again until the next bar code is reached and the robot starts reading again.

7.2.2 SCCharts Model of Bar Code Reader

In Appendix A the textual SCCharts model of this bar code reader is given. Figure **??** shows visualized model.

The initial state of the model is the Calibrate state. With a touch event the high and the low value for the light sensor is set.

The next state is the StartDelimiter state. A sequence of dark, light, dark is then needed to reach the final state. The state AdjustClock is dedicated to adjustments. In the real world the robot does not drive exactly at the specified speed and it is impossible that it always approaches the bar code in a 90° angle. This state evens these imprecisions and is entered after the start delimiter and after every digit that is read. At this moment it can be foreseen that a light line follows a dark line so the internal clock, that count the time it takes for reading one line of the bar code, is reseted.

In ReadDigit, the seven bar code lines are read and evaluated. The four digits that are read, are saved in an array and the state EndDelimiter is entered. This state ensures that the end delimiter sequence is read and additionally the digits are used to compute the actual number. The parity bit is also checked. In case the result is not a valid representation, the robot does not turn. It then reenters the StartDelimiter state to wait for the next bar code. This is the same behavior for an invalid sequence or a problem occurring during reading of the bar code.

The last operation needed is the rotation itself. The state Rotate handles this rotation. The result is used as rotation angle and, as described above, a negative value represents a counterclockwise rotation and a positive value a clockwise rotation. If the touch sensor is pressed during this rotation, the robot enters its final state and stops execution.

7.2.3 Possible Problems

In the following, possible problems during the development of the Mindstorm are stated. Afterwards, possible problem solving strategies are described with a focus on how the KIELER debugger can be used to identify the problem.

Three main problem sources are contemplated. The first one considers the start delimiter. The robot does not seem to recognize it and the reason must be identified. The second source for a problem is the reading of the digits. Sequences with light, light,

```
1 ! reset;
2 forwardCounter
3 % Output:
4 ;
5 forwardCounter
6 % Output:
7 ;
8 forwardCounter
9 % Output:
10 ;
11 touch high forwardCounter
12 % Output:
13 ;
14 forwardCounter
15 % Output:
16 ;
17 forwardCounter
18 % Output:
19 ;
20 forwardCounter
21 % Output:
22 ;
23 touch low forwardCounter
24 % Output:
25 ;
26 forwardCounter
27 % Output:
28 ;
29 forwardCounter
30 % Output:
31 ;
32 forwardCounter
33 % Output:
34 ;
35 forwardCounter
36 % Output:
37 ;
38 touch speedC speedB forwardCounter
39 % Output:
40 ;
41 speedC speedB resetH forwardCounter
42 % Output:
43 ;
44 speedC speedB resetH forwardCounter
45 % Output:
46 ;
47 speedC speedB resetH forwardCounter
48 % Output:
49 ;
50 speedC speedB resetH forwardCounter
51 % Output:
52 ;
53 resetS speedC speedB resetH halfSecond
       forwardCounter
54 % Output:
55 ;
56 resetS speedC speedB resetH forwardCounter
57 % Output:
58 ;
59 resetS speedC speedB resetH forwardCounter
60 % Output:
61 ;
62 resetS speedC speedB resetH forwardCounter
63 % Output:
64 ;
65 resetS speedC speedB resetH light forwardCounter
66 % Output:
67 ;
68 resetS speedC speedB resetH light forwardCounter
69 % Output:
70 ;
71 resetS speedC speedB second resetH light
       forwardCounter
72 % Output:
73 ;
74 resetS speedC speedB resetH light forwardCounter
75 % Output:
76 ;
77 resetS speedC speedB resetH light forwardCounter
78 % Output:
79 ;
80 resetS speedC speedB second resetH
       forwardCounter
81 % Output:
82 ;
83 resetS speedC speedB resetH light forwardCounter
84 % Output:
85 ;
86 resetS speedC speedB resetH light forwardCounter
87 % Output:
88 ;
89 resetS speedC speedB resetH light halfSecond
       forwardCounter
90 % Output:
91 ;
92 resetS speedC speedB resetH light forwardCounter
93 % Output:
94 ;
```

Figure 7.4. First lines of the created .eso-file for the bar code reader

dark lines are not properly recognized. The last considered source for a problem is the following. After reading four digits, the state for recognizing the end delimiter is not entered. The four digits are already read but the end delimiter state is not entered. These three different problem sources can all be approached in different ways. One possible approach is chosen and its usability is evaluated. Another interesting factor is the number of ticks that could be skipped by using the debugger. KIEM shows the tick number at the top right.

The first problem occurred in StartDelimiter. There are many different ways of entering this state, therefore a breakpoint at the state itself is most useful. The simulation will pause when entering this state. Now the internal behavior can be observed. The inputs and outputs can be reviewed and the reason for taking or not taking a transition is visible. The execution prior to this state can be fast forwarded and only relevant model elements are reviewed. An .eso-file was created before starting the debugging procedure. Figure 7.4 show the first lines of the .eso-file. Fast forwarding to the specified breakpoint results in step 13. The execution manager shows the tick number at the top right. Depending on the used .eso-file, this value may vary or if the trace in the .eso-file never enters the breakpoint state, simulation may not stop at all. The step number here is only mentioned as an indicator for the usability of the fast forwarding and the time that is saved. The used trace represents the reading of 1233.

The second problem occurs in the interior of the state ReadDigit. Thus, a breakpoint can be specified more precisely than just at the state ReadDigit itself. The problem occurred after reading a code sequence of light, light and dark. A breakpoint at the state LLD would pause at the cause of the problem. After entering this state, the transitions taken can be reviewed and possible twists in the trigger of the transition can be identified. At this point the debugger fast forwarded to step 36 and then the specified breakpoint is reached.

The third problem is located somewhere between the states ReadDigit and EndDelimiter. A breakpoint at the outgoing termination transition is reasonable because this is where the problem interval starts. The transitions taken and the value of the counter and thus the number of already read digits can be reviewed. Also, the moment when a transition is taken that does not lead to the EndDelimiter can be identified quickly because the prior model is not stepped through manually. The simulation is in step 116 when the execution is paused.

7.2.4 Analysis of Usability

The problems stated above can be further analyzed and located with the use of the debugger. The model is paused at the defined moment and there are reasonable settings for state and for transition breakpoints. Also, the fast forwarding enables a vast amount

of time saving and allows the user to concentrate on the relevant model elements and not worry about the way to get during the debug process. The integration in KIEM is facilitated by the debug button that configures the schedule for the execution manager without the user having to understand all its detail.

However, there are still possible improvements. After the .eso-file is created, the usage is comfortable and straight forward but the creation itself is time-consuming. It is necessary to think about the needed trace before debugging and this trace must be reproduced in simulation. In case the requirements on the trace change, the .eso-file must be recreated. The .eso-file for debugging the described problems includes reading one barcode and reaching the Rotate state and has 494 lines. This corresponds to 165 ticks. The model must be stepped though manually at least once to create the .eso-file. With growing models and numerous inputs and outputs, this is a tedious task. The data table to set the inputs gets confusing when scrolling starts to be enabled because of the vast amount of inputs and outputs. Additionally, numerous .eso-files must be created to enable access to all model parts and this results in even more time consumption.

After setting the inputs for the next step, the step button itself must also be chosen by clicking on it. This shifts the focus for a moment and interrupts the concentration on the input setting.

Different randomly generated models have been tested. Breakpoint have been set at arbitrary positions and the correct behavior during the execution was verified.

Chapter 8

Conclusion

This chapter summarizes the results of this thesis and the main functionalities of the newly introduced elements of the debugger. The requirements are compared with the implementation and the evaluation results.

Additionally, this thesis detected many new approaches that can be implemented for future additional or alternative implementation and research. This future work is presented in Section 8.2.

8.1 Summary

In this thesis, different debugging possibilities are explored. This marks the first part of the thesis. Their suitability in the context of state machines is studied and the overall benefit for error finding is estimated. These different options differ greatly in their approach and there are still many ways how debugging could be facilitated. The integration of documentation would facilitate the understanding of SCCharts elements for unexperienced users. Furthermore, changes in model semantics are easily accessible.

Additionally, a static analysis with feedback for the user can prevent situations where compilation fails. With a model that does not compile, also the introduced debugger cannot be used. Therefore compilation errors can be prevented and after compiling the debugger can be used for detecting the cause for an error.

The second contribution of this thesis is the debugger for KIELER. It is embedded in the KIEM context and benefits from features resulting of its use. Newly introduced features of this debugger are the specification of breakpoints in the editor, their visualization in the editor, their visualization in the SCCharts diagram view and the interaction with the simulation. The breakpoints and the debugger elements are implemented using Eclipse features and the image for the icons resembles the icon images used in Eclipse applications.

Activating the debug mode results in adding the debugger component and the KART component to the current execution schedule. In this mode in each step, all active model elements are checked for breakpoints. In case an active model element is marked with a breakpoint the simulation is paused without further user interference. Note that this pausing is only perceptible when simulation is executed in run mode.

In addition to the debug mode, the fast forward mode can be chosen. This will cause the model simulation to execute the model with the information given in the .eso-files until a breakpoint is reached. This execution is performed with as little time delay as possible.

The environment needs to be modeled by the use of an .eso-file. This is necessary because input information must be accessible especially when fast forwarding. The usage of .eso-files is enabled through the integration of KART as one of the data components in the execution manager. This component loads the .eso-files and reads them.

8.2 Future Work

This thesis presents many approaches that can be implemented as future work. Especially in Chapter 3 other debugging approaches are given.

8.2.1 General Future Work

The integration of documentation in KIELER is one of the approaches that are discussed in Chapter 3. This can be realized through the Eclipse content assist or the Help feature. The documentation in the help feature would have a static appearance. It would be viewable as a browsable book but in the content assist this documentation is more dynamic. It can be compared to `Javadoc` documentation.

Static analysis is another approach. The written code is analyzed and compiler errors could be predicted. Appropriate feedback should then be given to the user so they are able to fix the bugs and the model can be compiled. The dead code analysis could also be extended so non-reachable code is identified and the model complexity is potentially reduced.

8.2.2 Optimizations

The introduced debugger brings many features with it, but there are certain features that can be improved and optimized.

The specification of breakpoints is currently only possible in the editor. The breakpoint is then visualized in editor and diagram view. The opposite could also be implemented. In that case, specifying a breakpoint in the diagram view would be possible through double clicking or maybe right-clicking. This would cause the same amount of control offered on the visual and on the textual side.

Also the different possible breakpoints can be extended. At the moment states and transitions correspond to a valid breakpoint but also regions or other model elements

might be reasonable. Additionally, in the data flow context a representation should be thought of so debugging features can be enabled there as well.

The specification of the input currently works with .eso-files. This can also be improved because especially for large models the creation of these files is a tedious task. An approach similar to the model input actors in the open-source framework Ptolemy [Lee03] could be taken. Ptolemy offers a actor-oriented design for different model semantics and inputs can be provided as a sequence of values. They can be changed individually and thus the specification of input values is modularized.

Alternatively, an input interval could be specified that sets an input every seventh tick for example. A shortcut for pressing the step button on the keyboard would also be helpful because it enables keeping the focus on the data table and the signals view during the creation of an .eso-file. By having to press the button with the mouse the focus is slightly shifted in every tick.

In addition, a partial simulation of the model can be implemented. The desired state must be specified somehow and then this can be simulated without its surrounding model elements. When it is verified that its inner elements work properly the next larger model state can be chosen and simulated. This possibly reduces relevant inputs for the inner testing and the model can be checked from the inside by first verifying that the inner states behave as desired. Note that simulating non-hierarchical states is not reasonable because they have no inner behavior. The needed inputs, outputs and variables must be filtered for the inner model, but then model parts can be checked in a modular manner.

The debugger uses KIEM as a simulation framework. Therefore, the behavior of actual models, like a LEGO Mindstorm robot, is only simulated. A more accurate debugging feature would operate directly with the execution of the debugger. Actual inputs could then be provided and problem only occurring in the real environment can be debugged.

Lastly, checking on the existence of a breakpoint can be improved by using tracing [Sch14]. This would enable the breakpoint visualization to be followed through the transformation procedure. Then, the evolvement of different model elements would be visible and the consequences for the simulation and model execution become clearer. Additionally, the breakpoints could be traced into the `C` code generation and the GDB could be used to enable pausing in between micro steps.

A different approach to enable micro stepping could be implemented similarly to the approach for SJ [Hei10]. The instruction view lists the different micro steps that the macro step consisted of. The linear order corresponds to their execution order. Note that the macro step is still performed entirely but the execution of the micro steps can be stepped back. The micro steps that are already executed are marked green and the ones that are not, are marked red. Therefore a breakpoint at a micro step can be realized by performing the macro tick and stepping back to the specific micro step in the instruction view, where execution was supposed to pause. Figure 8.1 shows the appearance of the instruction

view for the SJ context. The different micro steps are listed in the SJ Instruction View and the code lines are highlighted according to the already performed micro steps. The appearance of the view is close to the KIEM view except for this specifying micro steps. Step, run and stop buttons are provided.

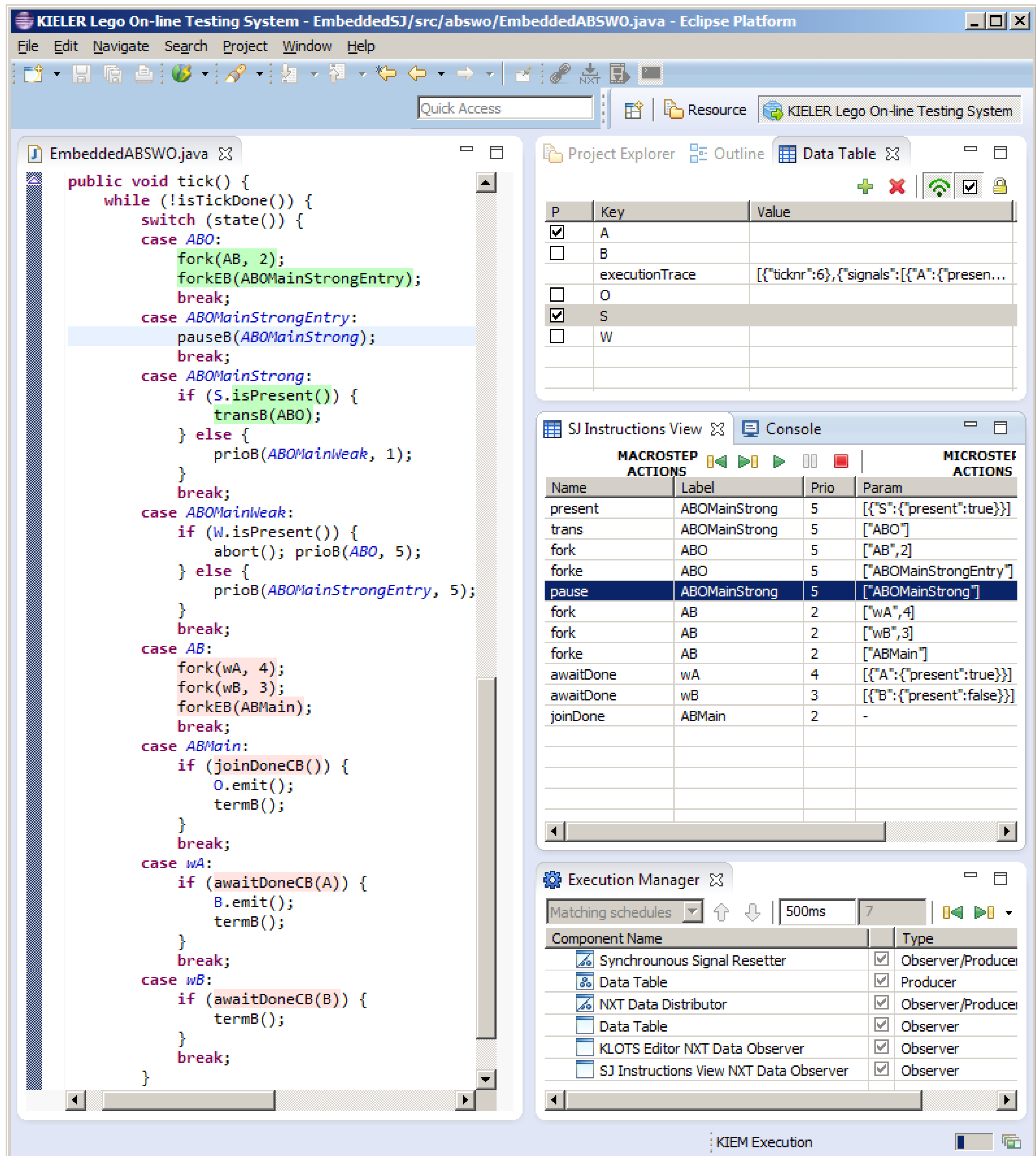


Figure 8.1. SJ! Instruction View as presented in [MHH13]

Appendix A

Textual Description of Bar Code Reader

```
// ++++++++++++++++++++++++++++++++ MINDSTORM PREPARATION ++++++++++++++++++++++++ //
//                                                           //
//                  MotorSpeed MotorSpeed                    //
//  |---------| A |---------| B |---------| C |---------| |---------| //
//  |                                                  |      //
//  |---------| S1 |---------| S2 |---------| S3 |---------| S4 |---------| //
//                 TouchSensor LightSensor                   //
//                                                           //
// ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++ //

scchart Barcode {

 // -------------------------------- USED WRAPPER --------------------------------- //
 @Wrapper Clock, "100"
 input bool second;

 @Wrapper ResetClock, second
 output bool resetS;

 @Wrapper Clock, "50"
 input bool halfSecond;

 @Wrapper ResetClock, halfSecond
 output bool resetH;

 @Wrapper Clock, "7"
 input bool turnClock;

 @Wrapper ResetClock, turnClock
 output bool resetTurn;


 @Wrapper MotorSpeed, B
 output int speedB;
 @Wrapper MotorSpeed, C
```

```
36  output int speedC;
37
38  @Wrapper LightSensor, S3
39  input bool light;
40
41  @Wrapper CalibrateHigh, S3
42  output bool high;
43
44  @Wrapper CalibrateLow, S3
45  output bool low;
46
47  @Wrapper TouchSensor, S2
48  input bool touch;
49
50
51  // --------------------------------- USED VARIABLES ---------------------------- //
52  // Saves the four digits in an array.
53  int numbers[4];
54
55  // This counter is incremented when a new digit is read.
56  output int counter = 0;
57
58  // Invalid Barcode.
59  output bool error = false;
60
61  // Used for rotation.
62  output int turnCounter = 0;
63
64  // Adjustments so robot starts rotating with the axis on the line, not the light sensor.
65  output int forwardCounter = 14;
66
67
68  // ---------------------------------- OUTPUTS --------------------------------- //
69  output bool valid = false;
70
71  output int result = 0;
72
73
74  // ********************************** CALIBRATE ********************************** //
75  initial state Calibrate {
76
77    initial state Wait
78    --> CalibrateBlack with touch /high = true;
79
80    state CalibrateBlack
81    --> DoneBlack with !touch /high = false;
82
```

```
83    state DoneBlack
84    --> CalibrateWhite with touch /low = true;
85
86    state CalibrateWhite
87    --> Done with !touch /low = false;
88
89    final state Done;
90  }
91  >-> StartDelimiter with touch/speedB = 100; speedC = 100;
92
93  // ********************************* START DELIMITER ****************************** //
94  state StartDelimiter {
95
96    entry /result = 0; counter = 0; valid = false;
97
98    initial state Read
99    --> AimLineMiddle with !light /resetH = true; error = false;
100
101   state AimLineMiddle
102   --> StartD with !light && halfSecond /resetS = true
103   --> Read with light && halfSecond;
104
105   state StartD
106   --> StartDL with light && second;
107
108   state StartDL
109   --> ReceivedStartDelimiter with !light && second
110   --> Read with light && second;
111
112   final state ReceivedStartDelimiter;
113 }
114 >-> AdjustClock;
115
116 // ***************************** ADJUSTCLOCK_PRE *********************************** //
117 state AdjustClock_pre
118 --> StartDelimiter with error
119 --> AdjustClock;
120
121 // ****************************** ADJUST CLOCK ************************************* //
122 state AdjustClock {
123
124   initial state Step1
125   --> Step2 with light /resetH = true
126   --> Done with !light && second /error = true;
127
128   state Step2
129   --> Done with halfSecond /resetS = true;
```

```
130
131      final state Done;
132    }
133    >-> Next;
134
135    // ********************************** NEXT ***************************************** //
136    state Next
137    --> StartDelimiter with error
138    --> ReadDigit with counter < 4
139    --> EndDelimiter with counter >= 4;
140
141    // ****************************** RECOGNIZE NUMBER ********************************** //
142    state ReadDigit {
143
144      initial state L
145      --> LL with light && second
146      --> LD with !light && second;
147
148      state LL
149      --> LLL with light && second
150      --> LLD with !light && second;
151
152      state LD
153      --> LDL with light && second
154      --> LDD with !light && second;
155
156      state LLL
157      --> Dark with light && second /error = true
158      --> LLLD with !light && second;
159
160      state LLD
161      --> LLDL with light && second
162      --> LLDD with !light && second;
163
164      state LDL
165      --> LDLL with light && second
166      --> LDLD with !light && second;
167
168      state LDD
169      --> LDDL with light && second
170      --> LDDD with !light && second;
171
172      state LLLD
173      --> LLLDL with light && second
174      --> LLLDD with !light && second;
175
176      state LLDL
```

```
177     --> LLDLL with light && second
178     --> Dark with !light && second /error = true;
179
180     state LLDD
181     --> LLDDL with light && second
182     --> Dark with !light && second /error = true;
183
184     state LDLL
185     --> LDLLL with light && second
186     --> Dark with !light && second /error = true;
187
188     state LDLD
189     --> Dark with light && second /error = true
190     --> LDLDD with !light && second;
191
192     state LDDL
193     --> LDDLL with light && second
194     --> LDDLD with !light && second;
195
196     state LDDD
197     --> LDDDL with light && second
198     --> LDDDD with !light && second;
199
200
201     state LLLDL
202     --> Something with second /numbers[counter] = 9;
203
204     state LLLDD
205     --> Something with second /numbers[counter] = 0;
206
207     state LLDLL
208     --> Something with second /numbers[counter] = 2;
209
210     state LLDDL
211     --> Something with second /numbers[counter] = 1;
212
213     state LDLLL
214     --> Something with second /numbers[counter] = 4;
215
216     state LDLDD
217     --> Something with second /numbers[counter] = 6;
218
219     state LDDLL
220     --> Something with second /numbers[counter] = 5;
221
222     state LDDLD
223     --> Something with second /numbers[counter] = 8;
```

```
224
225    state LDDDL
226    --> Something with second /numbers[counter] = 7;
227
228    state LDDDD
229    --> Something with second /numbers[counter] = 3;
230
231    state Something
232    --> Dark with !light && second /counter++
233    --> Dark with light && second /error = true;
234
235    final state Dark;
236  }
237  >-> AdjustClock_pre;
238
239  // ****************************** STOP DELIMITER ********************************** //
240  state EndDelimiter {
241
242    initial state EndL
243    --> ReceivedStopDelimiter with light && second /error = true
244    --> EndLD with !light && second /result = numbers[0]*100 + numbers[1]*10 + numbers[2];
245
246    state EndLD
247    --> ReceivedStopDelimiter with !light && second /error = true
248    --> EndLDL with light && second && result >= 500 /result = result - 1000; valid = (9 -
           ((numbers[0] + numbers[1] + numbers[2]) % 10) == numbers[3])
249    --> EndLDL with light && second /valid = (9 - ((numbers[0] + numbers[1] + numbers[2]) %
            10) == numbers[3]);
250
251    state EndLDL
252    --> ReceivedStopDelimiter with light && second /error = true
253    --> ReceivedStopDelimiter;
254
255    final state ReceivedStopDelimiter;
256  }
257  >-> WasError;
258
259  // ************************************** WAS ERROR ****************************** //
260  state WasError
261  --> StartDelimiter with error
262  --> Rotate with valid
263  --> StartDelimiter;
264
265  // ************************************* ROTATE *********************************** //
266  state Rotate {
267    initial state KeepDriving
268    --> KeepDriving with forwardCounter > 0 && second /forwardCounter--
```

```
269     --> Init with forwardCounter == 0 /forwardCounter = 14;
270
271     state Init
272     --> Wait with result > 0 /speedB = -100; speedC = 100; turnCounter = 2 * result;
            resetTurn = true
273     --> Wait with result < 0 /speedB = 100; speedC = -100; turnCounter = 2 * (-result);
            resetTurn = true
274     --> Wait with result == 0;
275
276     state Wait
277     --> Wait with turnCounter > 0 && turnClock /turnCounter--
278     --> Done with turnCounter == 0 && turnClock /speedB = 100; speedC = 100;
279
280     final state Done;
281   }
282   --> End with touch /speedB = 0; speedC = 0
283   >-> StartDelimiter with /error = false; counter = 0; result = 0;
284
285   // ************************************* END *************************************** //
286   final state End;
287
288 }
```

Appendix B

User Manual

This chapter addresses to end users. It includes the documentation for the KIELER SCCharts debugger and useful advices.

B.1 Requirements

In order to use the debugger, different steps should be taken to benefit most from it. In the following, these steps are listed and a short summary of the needed knowledge about KIELER is given.

Create a SCCharts model that is supposed to be tested The first step is modeling. It is recommended to use the KIELER Modeling Perspective. The SCCharts is described in .sct-language and the diagram view synthesizes a visualization. At this point different elements of SCCharts should be well-understood in order to use them appropriately. Compiling regularly helps to identify compiler errors as early as possible. At the end of the modeling process, check on the correct use of immediate transitions and the existence of final states, where they are needed.

Get familiar with KIEM At this point the model is done. You think that it behaves the way it is supposed to and now its time to check if it does. Get familiar with the KIEM simulator. The appropriate perspective is now the KIELER Simulation Perspective. The Execution Manager is placed at the bottom, arranged centrally. If you simply want to simulate the model to check on its behavior, the predefined schedule must not be changed. At the top right you will see your inputs and outputs as a time line. Below is a table, where the inputs and output can be set.

Be able to simulate your model with KIEM After getting familiar with the KIEM setup, try to simulate your model. You either have the possibility to step tick by tick or you chose the run mode. This will take the time specified for one tick and then go on automatically. Set some inputs, review the outputs.

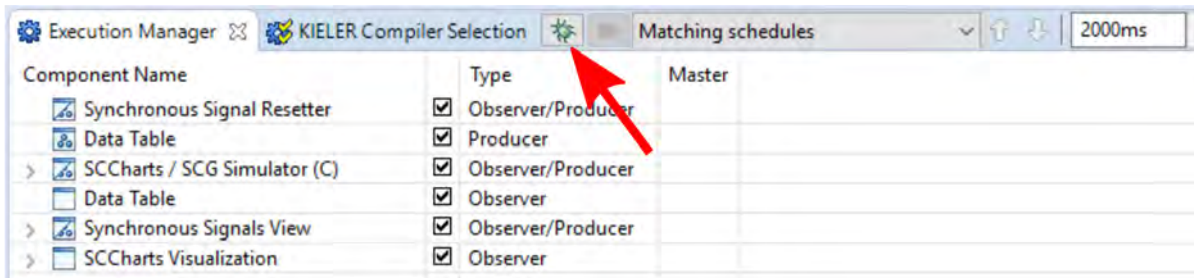
Create an .eso-file containing traces for the model Assuming you found a bug during simulation, the next step in order to use the debugger are .eso-files. You need them to have a source for the model inputs. First, simulate your model. Set inputs properly, so a trace is created that passes relevant model elements. It is important that the error happens during simulation of that trace, because this trace will be used during the debug process. In case your model does not need inputs, this step can be skipped.

Think of positions for breakpoints The next step is to specify appropriate breakpoints. They should be placed prior to the moment the error occurred. Note that the model is executed up until this breakpoint *including* the model element itself. A transition with a breakpoint for example is taken before simulation is stopped.
Breakpoints can be specified at states or transitions by double-clicking on the ruler. They are visualized in the editor and the diagram view. When specifying a breakpoint think about the .eso-traces again. Make sure that the model elements are run through when simulating with that trace, otherwise the model will not pause itself. Now the prerequisites for the debugger are set and it can be used. The next section describes the scope of the debugger and its usage.

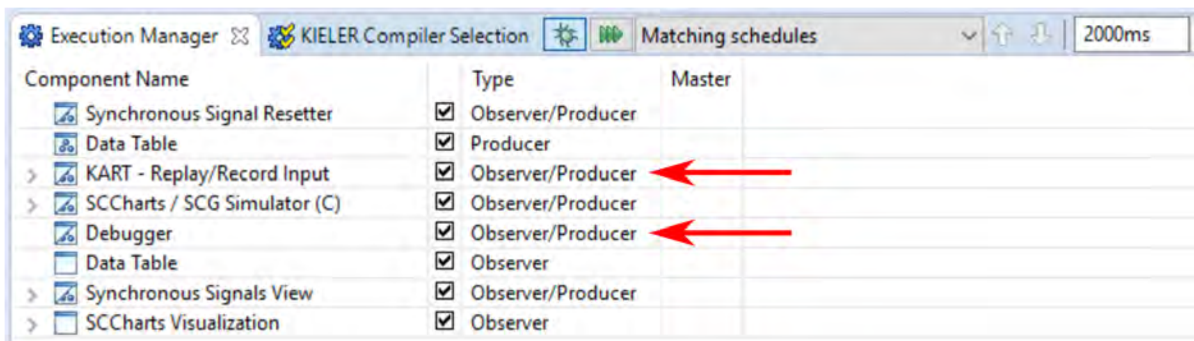
B.2 KIELER SCCharts Debugger

The first step towards debugging is activating the debug mode. In Figure B.1(a), the button that activates the debug mode is pointed out with an arrow. As a consequence, the KIEM KART component and the debugger component are added to the current schedule. Figure B.1(b) shows where these components are added in the schedule.

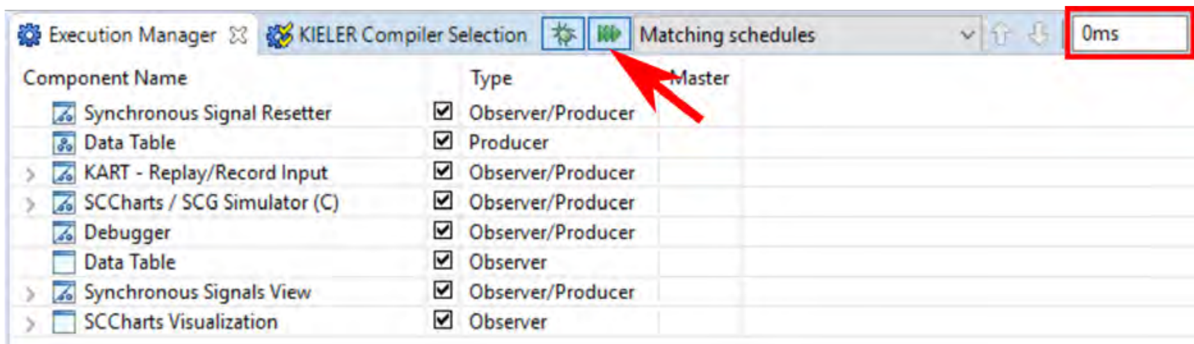
Now, the execution can be started. In run mode the execution pauses at breakpoints. The possibility to fast forward model parts that do not contain breakpoints, is still given. The fast forward button is still enabled after the execution is started, other than the debug button. In Figure B.1(c) it is shown that activating fast forward mode will set the average cycle time to zero. This will cause the execution to have as little time delay as possible for every step.



(a) Button to activate the debug mode



(b) Components added after activating debug mode



(c) Average step time after activating fast forwarding

Figure B.1. Summary of the button functionalities of the debugger

Appendix C

Acronyms

GDB Gnu Project Debugger

JDT Java Development Tooling

JPDA Java Platform Debugger Architecture

KART KIELER Automated Regression Testing

KIELER Kiel Integrated Environment for Layout Eclipse Rich Client

KIEM KIELER Execution Manager

LTL Linear Temporal Logic

MoC Model of Computation

PDE Plugin Developer Environment

SAT Satisfiability

SC MoC Sequentially Constructive Model of Computation

SCChart Sequentially Constructive Chart

SCG Sequentially Constructive Graph

SMV Symbolic Model Verifier

SJ Synchronous Java

VM Virtual Machine

SWT Standard Widget Toolkit

Bibliography

[06] *SCADE Technical Manual*. 5.1. Esterel Technologies. Feb. 2006.

[AMH+14] Joaquín Aguado, Michael Mendler, Reinhard von Hanxleden, and Insa Fuhrmann. “Grounding synchronous deterministic concurrency in sequential programming”. In: *Proceedings of the 23rd European Symposium on Programming (ESOP’14), LNCS 8410*. Grenoble, France: Springer, Apr. 2014, pp. 229–248.

[And95] Charles André. *Synccharts: a visual representation of reactive behaviors*. Tech. rep. Oct. 1995.

[BC09] Paul Butcher and Jacquelyn Carter. *Debug it! - find, repair, and prevent bugs in your code*. Raleigh, Noth Caolina and Dallas, Texas: Pragmatic Bookshelf, 2009. ISBN: 978-1-934-35628-9.

[BCE+03] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. “The Synchronous Languages Twelve Years Later”. In: *Proc. IEEE, Special Issue on Embedded Systems*. Vol. 91. Piscataway, NJ, USA: IEEE, Jan. 2003, pp. 64–83.

[Ber00] Gérard Berry. “The foundations of Esterel”. In: *Proof, Language, and Interaction: Essays in Honour of Robin Milner*. Ed. by Gordon Plotkin, Colin Stirling, and Mads Tofte. Cambridge, MA, USA: MIT Press, 2000, pp. 425–454. ISBN: 0-262-16188-5.

[Bry86] Randal E Bryant. “Graph-based algorithms for boolean function manipulation”. In: *IEEE Transactions on computers* 100.8 (1986), pp. 677–691.

[CKL04] Edmund Clarke, Daniel Kroening, and Flavio Lerda. “A tool for checking ansi-c programs”. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2004, pp. 168–176.

[GGB+91] Paul Le Guernic, Thierry Goutier, Michel Le Borgne, and Claude Le Maire. “Programming real time applications with SIGNAL”. In: *Proceedings of the IEEE* 79.9 (Sept. 1991), pp. 1321–1336.

[Han99] Per Brinch Hansen. “Java’s insecure parallelism”. In: *SIGPLAN Not.* 34.4 (Apr. 1999), pp. 38–45. ISSN: 0362-1340.

[Har87] David Harel. “Statecharts: A visual formalism for complex systems”. In: *Science of Computer Programming* 8.3 (June 1987), pp. 231–274.

[HCR+91] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. “The synchronous data-flow programming language LUSTRE”. In: *Proceedings of the IEEE* 79.9 (Sept. 1991), pp. 1305–1320.

[HDM+14a] 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)*. Edinburgh, UK: ACM, June 2014.

[HDM+14b] 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)*. Long version: Technical Report 1311, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, December 2013, ISSN 2192-6274. Edinburgh, UK: ACM, June 2014.

[Hei10] Mirko Heinold. *Synchronous Java*. Bachelor thesis, Kiel University, Department of Computer Science. Sept. 2010.

[HFS11] Reinhard von Hanxleden, Hauke Fuhrmann, and Miro Spönemann. “KIELER—The KIEL Integrated Environment for Layout Eclipse Rich Client”. In: *Proceedings of the Design, Automation and Test in Europe University Booth (DATE’11)*. Grenoble, France, Mar. 2011.

[HK99] Wolfgang A Halang and Rudolf Konakovsky. *Sicherheitsgerichtete echtzeitsysteme*. München: Oldenbourg Industrieverlag, 1999. ISBN: 978-3-486-24036-8.

[HMA+13a] Reinhard von Hanxleden, Michael Mendler, Joaquín Aguado, Björn Duderstadt, Insa Fuhrmann, Christian Motika, Stephen Mercer, and Owen O’Brien. “Sequentially Constructive Concurrency—A conservative extension of the synchronous model of computation”. In: *Proc. Design, Automation and Test in Europe Conference (DATE’13)*. Grenoble, France: IEEE, Mar. 2013, pp. 581–586.

[HMA+13b] Reinhard von Hanxleden, Michael Mendler, Joaquín Aguado, Björn Duderstadt, Insa Fuhrmann, Christian Motika, Stephen Mercer, and Owen O’Brien. “Sequentially Constructive Concurrency—A conservative extension of the synchronous model of computation”. In: *Proc. Design, Automation and Test in Europe Conference (DATE’13)*. Long version: Technical

Report 1308, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, August 2013, ISSN 2192-6247. Grenoble, France: IEEE, Mar. 2013, pp. 581–586.

[HMA+13c] Reinhard von Hanxleden, Michael Mendler, Joaquín Aguado, Björn Duderstadt, Insa Fuhrmann, Christian Motika, Stephen Mercer, Owen O'Brien, and Partha Roop. *Sequentially Constructive Concurrency—A conservative extension of the synchronous model of computation*. Technical Report 1308. ISSN 2192-6247. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Aug. 2013.

[KM08] Fred Kröger and Stephan Merz. *Temporal logic and state systems*. Berlin Heidelberg: Springer-Verlag Berlin Heidelberg, 2008. ISBN: 978-3-540-67401-6.

[KY03] Hoon-Joon Kouh and Weon-Hee Yoo. "The efficient debugging system for locating logical errors in java programs". In: *International Conference on Computational Science and Its Applications*. Springer. 2003, pp. 684–693.

[Lee03] Edward A. Lee. *Overview of the Ptolemy project*. Technical Memorandum UCB/ERL M03/25. University of California, Berkeley, CA, 94720, USA, July 2003.

[Lee06] Edward A. Lee. "The problem with threads". In: *IEEE Computer* 39.5 (2006), pp. 33–42.

[LS11] Edward A. Lee and Sanjit A. Seshia. *Introduction to embedded systems, a cyber-physical systems approach*. Lulu, 2011. ISBN: 978-0-557-70857-4. URL: http://LeeSeshia.org.

[MFH09] Christian Motika, Hauke Fuhrmann, and Reinhard von Hanxleden. *Semantics and execution of domain specific models*. Technical Report 0923. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Dec. 2009.

[MH89] Charles E McDowell and David P Helmbold. "Debugging concurrent programs". In: *ACM Computing Surveys (CSUR)* 21.4 (1989), pp. 593–622.

[MHH13] Christian Motika, Reinhard von Hanxleden, and Mirko Heinold. "Programming deterministice reactive systems with Synchronous Java (invited paper)". In: *Proceedings of the 9th Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS 2013)*. IEEE Proceedings. Paderborn, Germany, 17/18 06 2013.

[Mot09] Christian Motika. "Semantics and execution of domain specific models—KlePto and an execution framework". http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/cmot-dt.pdf. Diploma thesis. Kiel University, Department of Computer Science, Dec. 2009.

[Mot16] Christian Motika. "Interactive incremental compilation for sequentially constructive charts (sccharts)". Dissertation. Kiel: Christian-Albrechts-Universität zu Kiel, Faculty of Engineering, 2016.

[MZ09] Sharad Malik and Lintao Zhang. "Boolean satisfiability from theoretical hardness to practical success". In: *Communications of the ACM* 52.8 (2009), pp. 76–82.

[OS02] Rainer Oechsle and Thomas Schmitt. "Javavis: automatic program visualization with object and sequence diagrams using the java debug interface (jdi)". In: *Software visualization*. Springer, 2002, pp. 176–190.

[PO11] Chris Parnin and Alessandro Orso. "Are automated debugging techniques actually helping programmers?" In: *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ACM. 2011, pp. 199–209.

[QM12] Jürgen Quade and Michael Mächtel. *Moderne realzeitsysteme kompakt - eine einführung mit embedded linux*. Heidelberg: dpunkt.verlag, 2012. ISBN: 978-3-864-91218-4.

[Sch09] Klaus Schneider. *The synchronous programming language Quartz*. Internal Report 375. Kaiserslautern, Germany: Department of Computer Science, University of Kaiserslautern, Dec. 2009.

[Sch14] Alexander Schulz-Rosengarten. "Framework zum Tracing von EMF— Modelltransformationen". http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/als-bt.pdf. Bachelor thesis. Kiel University, Department of Computer Science, Mar. 2014.

[SMS+15] Steven Smyth, Christian Motika, Alexander Schulz-Rosengarten, Nis Boerge Wechselberg, Carsten Sprung, and Reinhard von Hanxleden. *SC-Charts: the railway project report*. Technical Report 1510. ISSN 2192-6247. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Aug. 2015.

[Smy13] Steven Smyth. "Code generation for sequential constructiveness". http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/ssm-dt.pdf. Diploma thesis. Kiel University, Department of Computer Science, July 2013.

[The06] The model railway. *Project homepage*. Group of Real-Time and Embedded Systems, Department of Computer Science, Kiel, Germany. 2006. URL: http://www.informatik.uni-kiel.de/~railway.

[Ves85] Iris Vessey. “Expertise in debugging computer programs: a process analysis”. In: *International Journal of Man-Machine Studies* 23.5 (1985), pp. 459–494.

[XY03] Min Xie and Bo Yang. “A study of the effect of imperfect debugging on software development cost”. In: *IEEE Transactions on Software Engineering* 29.5 (2003), pp. 471–473.