Model-Based Debugging

Philip Eumann
stu121235@mail.uni-kiel.de

Master's Thesis
June 2020

Embedded and Real-Time Systems Group
Department of Computer Science
Kiel University

Advised by
M.Sc. Alexander Schulz-Rosengarten
Dipl-Inf. Steven Smyth
Prof. Dr. Reinhard von Hanxleden

Selbstständigkeitserklärung

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

Kiel,

Abstract

Source-level debugging is a standard procedure in software development, increasing productivity and software quality. For embedded safety-critical applications, where *model-based languages* are commonly used, high quality of software is particularly important. However, standard debuggers will either simulate the model-based code on its own, disregarding any host code surrounding it in its real environment, or run on the generated code like on a generic program, losing options to examine the source model's state.

To make debugging of such projects easier, this thesis proposes concepts for *model-based debugging* of generated code. A model-based debugger runs on the host language level and uses knowledge about the code generation process and additional marker comments in the generated code. With this information, it can visualize the generated code's memory state on the model level, making it easier for the user to grasp the current state of the model. Additionally, the debugger allows for the placement of host-language breakpoints through the source model, letting the user interrupt the control flow of the generated code in the right place without any knowledge about its structure.

The presented concepts are demonstrated and evaluated using SCCharts, a synchronous state machine-based language. A prototype of a model-based SCCharts debugger has been created and two small studies were conducted to evaluate its useability and intuitiveness.

Acknowledgements

First of all, I would like to thank professor von Hanxleden for the opportunity to write this thesis and the feedback during its creation. I also thank my advisers Alexander Schulz-Rosengarten and Steven Smyth for answering all my questions on KIELER and for their helpful feedback throughout the thesis. Thanks to Nis Wechselberg, Hauke Fuhrmann and the entire team at Scheidt&Bachmann for their feedback and for taking the time to participate in my evaluation. Thanks to everyone who participated in the studies to help me evaluate my work, and to those who tried, but couldn't make it.

I also want to thank my family for always being there for me and supporting me at all times. Last, but not least, I want to thank Johanna, without whom I would not have made it.

Contents

1 Introduction **1**
1.1 Debugging Overview 1
1.1.1 Breakpoints 1
1.1.2 Displaying and Editing the Memory State 2
1.1.3 Step-by-Step execution 3
1.2 SCCharts as an Example Language 3
1.3 Problem Statement 3
1.4 Goals 5
1.4.1 Setting Breakpoints 5
1.4.2 Displaying Runtime Information 5
1.4.3 Stepping 5
1.4.4 Minimal Invasiveness 6
1.5 Outline 6

2 Related Work **7**
2.1 Synchronous Languages 7
2.1.1 Constructiveness and Signals 7
2.1.2 SCCharts and the Sequentially Constructive MoC 8
2.2 Model-Based Compilation Approaches 9
2.2.1 State-Based Compilation 10
2.2.2 Priority-Based Compilation 11
2.2.3 Netlist-Based Compilation 14
2.3 Debugging SCCharts 15
2.4 Formal Verification 16
2.5 Compiler Tracing and its Applications 17
2.5.1 Performance Debugging 17
2.5.2 Debugging Optimized Code 18
2.5.3 Transparent Debugging of Dynamically Optimized Code 19
2.6 Visual Debugging 19
2.7 Semi-Automatic Debugger Generation 20
2.8 Model-Level Debugging 21

3 Used Technologies **23**
3.1 The Eclipse Platform 23
3.1.1 Plugin Infrastructure and Extension Points 23
3.1.2 Java Development Toolkit 24
3.1.3 Debugging in Eclipse 24
3.2 KIELER 26
3.2.1 Modular Compilation Systems 27
3.2.2 Model Tracing 27

4 Design and Concept **29**
4.1 Breakpoint Semantics . . . 29
4.1.1 State Breakpoints . . . 29
4.1.2 Transition Breakpoints . . . 30
4.1.3 Other Breakpoints . . . 32
4.2 Markers in Generated Code . . . 32
4.3 Source Model Access . . . 33
4.4 User Interface . . . 33
4.4.1 Debug Diagram View . . . 34
4.4.2 Setting Breakpoints . . . 35
4.4.3 Visual Semantics . . . 36
4.5 Changes to the Compilation Chain . . . 38
4.5.1 Adding Debug Markers . . . 38
4.5.2 Template Adaptations . . . 39
4.6 Finding Breakpoint Locations . . . 40
4.6.1 State Breakpoints . . . 40
4.6.2 Transition Breakpoints . . . 41
4.7 Retrieving Runtime Information . . . 41
4.7.1 Active States . . . 42
4.7.2 Executing States . . . 43
4.7.3 Executing Transition . . . 43
4.8 Implementation Approach for Priority-Based Compilation . . . 44
4.8.1 Marker Comments . . . 44
4.8.2 Setting Breakpoints and Extracting Runtime Information . . . 45
4.9 Implementation Approach for Netlist-Based Compilation . . . 46
4.9.1 Marker Comments . . . 46
4.9.2 Setting Breakpoints . . . 47
4.9.3 Extracting Runtime Information . . . 48

5 Implementation **51**
5.1 Plugin Setup . . . 51
5.2 Modularity . . . 51
5.3 Changes to the Compilation Chain . . . 52
5.4 Listeners . . . 54
5.4.1 Breakpoint Listener . . . 54
5.4.2 Part Listener . . . 54
5.4.3 Resource Change Listener . . . 55
5.5 Diagram View and Highlighting . . . 56
5.5.1 Highlighting . . . 56
5.5.2 Synthesis Hooks . . . 57
5.6 Extracting Runtime Information . . . 57

6 Evaluation **59**
6.1 Evaluation with Professional Developers . . . 60
6.1.1 Integration into the Existing Workflow . . . 60
6.1.2 UI and Useability . . . 60
6.1.3 Stability and Performance . . . 61
6.1.4 Questions on Breakpoints . . . 62

6.1.5 Questions on Stepping 63
6.2 Evaluation with University Members 64
6.2.1 Study Setup 64
6.2.2 Quantitative Results 65
6.2.3 Qualitative Results 70
6.3 Evaluation Summary 74

7 Conclusion **75**
7.1 Summary 75
7.2 Future Work 77
7.2.1 Better Access to Runtime Variables 77
7.2.2 Improve Linking of Editor and Diagram View 78
7.2.3 Support for Other Compilation Approaches and Host Languages 78
7.2.4 Additional Information in the Diagram 78

A Debugging Cheatsheet **81**

B Questionnaire for Professional Developers **83**

C Online Survey for University Members **85**

Bibliography **93**

List of Figures

1.1 An example of Debugging in the Eclipse IDE. 2
1.2 An example SCChart within the development environment. 4

2.1 Example Esterel Programs. 8
2.2 An Example SCChart. 10
2.3 Code extracts generated with the state-based approach. 11
2.4 SCG of the ABRO example with assigned priorities. 13
2.5 Example code generated with priority- and netlist-based approaches. 14
2.6 SCCharts debugging components added by Lena Grimm [Gri16] 16
2.7 Example Esterel program with critical path highlighting as proposed by Ju et al. [JHR+08] 18
2.8 Visual debugging tool *Lens* as proposed by Mukherjea and Stasko [MS94]. 20
2.9 An example system as visualized by the model-level debugger proposed by Djukić et al. [DPL16]. 22

3.1 Debug perspective in Eclipse. 25
3.2 A screenshot of the KIELER SCCharts development perspective. 26
3.3 The KIELER compiler selection with the state-based compilation system and intermediate results. 27

4.1 Original and transformed model extract. 30
4.2 `AlarmSound` region from the `AlarmSystem` SCChart. 31
4.3 The demonstrator's user interface with components relevant to debugging highlighted. 34
4.4 Comparison of use cases for diagram views in KIELER. 35
4.5 Different types of breakpoints. 36
4.6 An SCChart during a debug run. 37
4.7 A transformed and annotated SCCharts model. 39
4.8 Steps to placing a breakpoint corresponding to a model element selected by the user. . 40
4.9 Code extracts from `TrafficLight` SCChart. 42
4.10 Steps to extracting executing model elements from the program's runtime memory. . . 43
4.11 A priority-based compilation example. 44
4.12 Code examples for marker placement in priority-based code. 45
4.13 Code examples for marker placement in netlist-based code. 47
4.14 Approaches to determining active and executing states in netlist-based code and their advantages and drawbacks. 49

5.1 Different components and their reusability across different languages and compilation approaches. 52
5.2 Source model excerpt and generated code. 53
5.3 Different listeners and their uses. 54
5.4 Process triggered by resource changes. 55
5.5 Model management and highlighting components. 56

6.1 Results concerning the integration into the existing workflow. 60
6.2 Results concerning UI and useability. 61
6.3 Results concerning stability and performance. 62
6.4 Results concerning the usage of present breakpoint types. 63
6.5 Results concerning the introduction of new breakpoint types. 63
6.6 Results concerning the stepping support of the debugger. 63
6.7 `AlarmSystem` SCChart used in the online study. 66
6.8 `AlarmSound` region of the `AlarmSystem` SCChart. 67
6.9 Number of participants from either group that found bug #1. 67
6.10 `SystemStatus` region of the `AlarmSystem` SCChart. 68
6.11 Number of participants from either group that found bug #2. 68
6.13 Number of participants from either group that found bug #3. 69
6.12 Extracts from `AlarmSystemEnvironment.java`. 69
6.14 Total time in minutes taken by participants. 70
6.15 Results of questions on model and bug reports. 71
6.16 Feedback from the control group on the debugging methods they used. 72
6.17 Feedback from the experimental group on the demonstrator. 73

7.1 KIELER simulation variables view. 77
7.2 Simulated SCChart with live values in the diagram. 79

C.1 Welcome page with instructions for KIELER installation and code import. 85
C.2 Group assignment page of the study. 86
C.3 Confirmation page to ensure that all preparations have been taken 86
C.4 Group-specific page with instructional video and links. 87

Chapter 1

Introduction

Many real-time systems have high requirements for safety and thus code correctness. While for some high-risk systems, such as power plant controls or aviation software, formal verification is commonly used today [BBD+17], testing remains the main way of finding faults in code for most applications [OR14]. Many faults become apparent and easy to fix once they are discovered, however, some are harder to locate and require extensive debugging. This becomes particularly difficult for heterogeneous systems where components are developed using different languages since in that case, no single debugger can be used for the whole system. The concept of *model-based design* usually results in such systems.

In model-based-design, an additional abstraction level, usually in form of a new *modelling language*, is added on top of a pre-existing general-purpose language, the *host language*. Much like other high-level languages, this addition does not bring more expressiveness since the new language is compiled into an equivalent program in the host language. However, the new modelling language usually makes developing certain types of software easier and quicker since less code needs to be written, helping manufacturers to reduce development time while increasing quality [DF07].

As mentioned above, this improvement comes at the cost of some difficulties during debugging. While debuggers for both the host and the model-based language may be available, projects will often consist of both code hand-written in the host language and code generated from a model. The model can then be used as a high-level control instance that orchestrates the system's behavior, while hand-written host language code handles low-level I/O and other tasks. The debugger for the host language works on both the generated and the hand-written code, but not on the original model, which leaves it up to the user to relate the generated code with the original model. On the other hand, the debugger for the model-based language can be used on the source model, but oftem does not take interactions with other parts of the project into consideration if they were written in the host language.

Therefore, this thesis presents concepts for debugging such heterogeneous projects with an augmented debugger for the host language. Such a debugger runs on the host-language code, but aids the user in relating the source model with the code generated from it.

1.1 Debugging Overview

To give a better idea of the expectations towards a debugger, this section will cover the basic underlying concepts of debugging.

1.1.1 Breakpoints

The main idea behind a debugger is to examine a program while it is running to understand its inner workings in a way that would not be possible simply by running it and observing the input / output behavior. However, most debuggers do not actually allow interactions while the program is running, since its state is changing too quickly in that case. Instead, they allow the user to interrupt

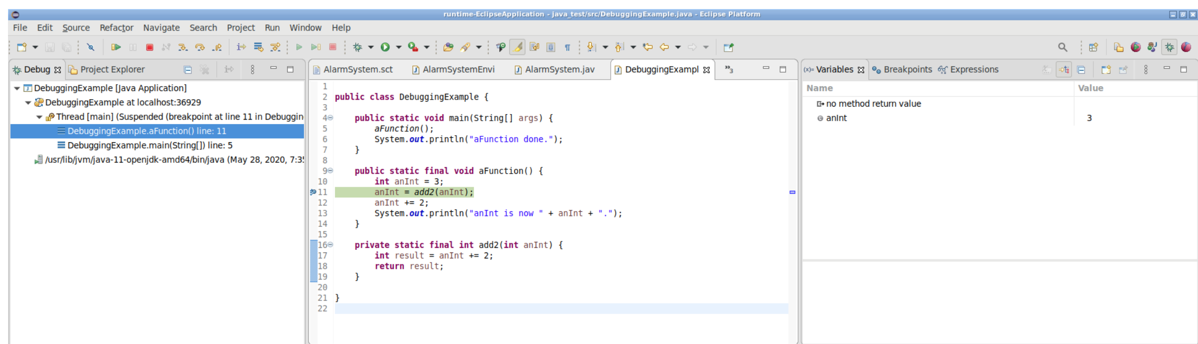


Figure 1.1. An example of Debugging in the Eclipse IDE.

the program's execution, effectively freezing the associated memory and allowing interaction with it before resuming.

A common concept to achieve such interruptions in a controlled way is that of *breakpoints*. Breakpoints are markers that are associated with a certain event in the program, e. g., the execution of a certain instruction or the occurrence of an exception. The program will run normally up to the occurrence of the specified event, which will then lead to a program suspension. After the user is done with examining the program state, they can then resume the program until the next breakpoint is hit or the execution terminates.

Depending on the language and debugger, the options for breakpoints and when they can be set may differ. For Java, the *Java Virtual Machine (JVM)* provides support for setting breakpoints at any time during execution, be the program suspended or running, if in debug mode. Figure 1.1 shows an example of an *Integrated Development Environment (IDE)* with Java debugging support. A breakpoint on line 11 caused the execution to suspend there.

Other languages may require a compilation with certain debug flags and breakpoints placed before compilation to allow proper suspension in the desired locations.

1.1.2 Displaying and Editing the Memory State

Once the execution has been suspended, the debugger must present the current state of the program to the user and possibly allow interaction with it. Most debuggers for imperative languages will allow the user to view the contents of the program's runtime memory, showing variable values, stack state or register values. Debuggers integrated in IDEs will usually highlight the line of code on which the program has been suspended, making it easy for the user to spot the current program state. In Figure 1.1, line 11 is highlighted in green since the program has been suspended there. The view to the left shows the current call stack of the suspended thread.

Once the user has seen the memory state, they may use appropriate inputs to influence the execution the way they need to to test certain behavior. However, since that may not always be possible, many debuggers will also allow direct interaction with the program memory while suspended. With such features, the user can simply set internal variables to their desired values even if that would not be possible in a regular program run. To the right of Figure 1.1, the variables in the current scope can be read and edited.

1.1.3 Step-by-Step execution

If the user wants to examine the program execution in small steps, placing a breakpoint on (almost) every line is tedious. Therefore, an important functionality of each debugger is to execute the code step by step without the use of breakpoints. Most debuggers support three stepping modes: *Step into*, *Step over* and *Step return* [LKV11]. Any step will suspend when a breakpoint is hit before the step ends.

The *Step into* command will also suspend at the first code location on the next lowest hierarchy level, the *Step over* command suspends when the next code location on the present hierarchy level is reached, and *Step return* only suspends when arriving at a code location on the next highest level. The exact concept of hierarchy levels and code locations depends on the language under analysis and can therefore not be generalized easily.

For imperative languages such as C or Java, each line of code is considered a code location. In the example program in Figure 1.1, `Step over` will therefore go to line 12 in the same method, `Step into` will go to line 17, the first line of the called method, and `Step return` will leave the current method and end on line 6 in the next highest method.

1.2 SCCharts as an Example Language

As a running example of a model-based language, *SCCharts* will be used throughout this thesis. SCCharts is a model-based synchronous language proposed by von Hanxleden et al. [HDM+13]. Since it has been designed for safety-critical applications, it is suitable for modelling complex control systems where the generated code is embedded into a host-language project. Therefore, the language suits as an example for this thesis. To illustrate and validate the presented approaches, I have implemented a demonstrator for SCCharts.

While SCCharts are a graphical, state-machine based language, a textual editor for easy editing is available, coupled with an automatically generated diagram representation. When the model is finished, it can be compiled to a host language such as C or Java using a variety of different compilation approaches. The language and its compilation are introduced in more detail in Section 2.1.

Figure 1.2 shows the SCCharts editor along with the automatically synthesized diagram next to it. The bottom view shows the compiler selection where different compilation approaches and target languages can be selected.

1.3 Problem Statement

As mentioned above, model-based programs can either be debugged directly with a specialized debugger for the modelling language or a more general one for the host language. Debugging the source model is useful for component testing and for finding logical errors in the model itself. However, detecting problems arising from the interactions between the model and its surrounding code can only be done through integration tests conducted with the full system, which is not possible using a debugger for the modelling language.

For this purpose, a debugger for the host language must be used to be able to observe both the hand-written and the generated code. Another reason to use a debugger on the generated code rather than the source model is that most code generators are not formally verified, therefore there is no guarantee that an error-free source model leads to error-free generated code. The topic of formal verification is covered in more detail in Section 2.4.

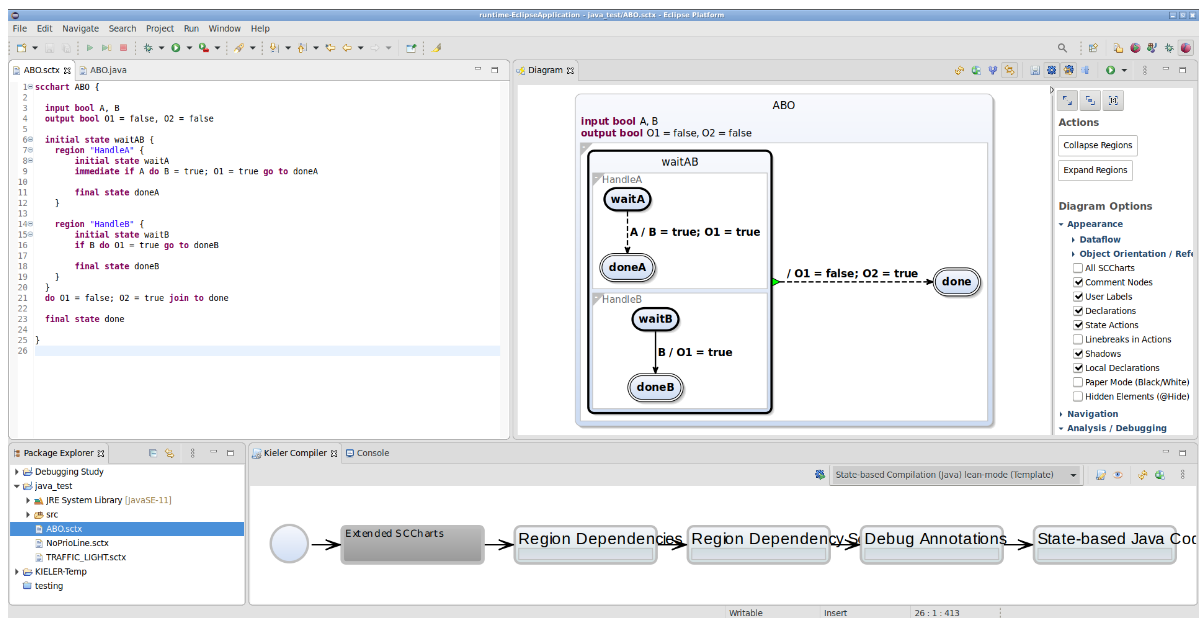


Figure 1.2. An example SCChart within the development environment.

However, the debugger for the host language only works on that language's abstraction level, not taking into consideration the overlying structure of the model. Depending on the compiler, there may be great structural differences between original and generated code. In some cases, the compiler focuses on performance, code size or similar metrics while not putting any emphasis on code readability. Even though there has been recent progress in generating more human-readable code from model-based languages [SMH18], the following issues remain when using host-language debuggers on generated code.

Navigating the generated code is hard. While the generated code is structured according to the source model, finding the code corresponding to certain model elements may be difficult. Navigating the source code efficiently requires detailed knowledge about the code generation process and the different transformations applied.

Placing breakpoints is non-trivial. If one wants to place breakpoints in the generated code, it may be difficult to find the appropriate code location(s). Depending on the model structure and compiler, *model-to-model (m2m)* transformations may be applied during the compilation progress, replacing complex source model elements with a set of multiple semantically simpler constructs. Without detailed knowledge of these transformations and the host language constructs they result in, it may be hard to impossible to find appropriate code locations for breakpoints.

Regular debugging tools do not offer enough information. When a breakpoint within the generated code is hit, it may be hard to find out what part of the model triggered the breakpoint. Even if one can determine the correct model element, this does not necessarily tell the user everything they need to know about the model's state. Even after finding the model element triggering the breakpoint, the values of I/O variables or internal states remain hidden from the user, again requiring knowledge about the code generation process to find the values within the generated code's runtime memory.

1.4 Goals

This thesis contributes concepts for debugging heterogeneous projects containing code generated from model-based languages. The main goal is to facilitate debugging within the hostcode environment to seamlessly integrate with the existing workflow and tooling for that language without confronting the user with any of the above challenges. To illustrate the concepts, a demonstrator implementation for SCCharts is developed as part of the thesis.

The following sections present different functionalities that such a debugger needs to provide. More details on the features and concepts for realizing them can be found in Chapter 4.

1.4.1 Setting Breakpoints

As mentioned above, finding the right place in the code for a breakpoint is non-trivial. Therefore, the tool needs to offer a mechanism to easily place breakpoints on model elements without the user needing to find the appropriate location(s) in the generated code. Depending on the language, this can either be done in a text editor by selecting the desired line of code or in a graphical view of the model for visual modelling languages.

When the user places a breakpoint on the source model element, the tool needs to automatically determine the appropriate code location(s) in the generated code and place a breakpoint in each one of them. This way, the execution of the generated code will suspend appropriately when the model element is reached. Note that the appropriate location for breakpoints depends on the breakpoint semantics; this topic will be discussed in more detail in Section 4.1.

1.4.2 Displaying Runtime Information

Section 1.3 stated that the information provided by the regular host language debuggers does not suffice for debugging the source model appropriately. Therefore, the tool needs to visually indicate the state of the source model whenever a breakpoint is hit.

There are many ways of doing this. Some approaches for visualizing the state of a program are discussed in Chapter 2. For visual languages, highlighting parts of the model is an option while many textual languages opt for highlighting the executing line of code along with tooltips for variables displaying their runtime values.

1.4.3 Stepping

There are two possible ways a debugger can support stepping through the generated code. Since the debugging of heterogeneous projects with both generated and hand-written host language code is the goal, the tool needs to support at least stepping on the host language level. This way, the user is able to step through their host language project as usual and when stepping into the generated code, the tool seamlessly displays the same information for the current code location as it would for a breakpoint in the same place. When stepping out of the generated code, the user can seamlessly return to debugging their hand-written code. In this variant, stepping would be supported on statement and method level in the generated code, but not on the original model level.

The second variant is to implement a custom debug model for the modelling language, allowing to interpret the different stepping commands with respect to the model and regardless of the unterlying host language. For this concept, the tool needs to translate each step on the model level into a step (or a sequence of steps) on the host language level. Even though this variant introduces more complexity

to the tool, it may be useful especially since it does not require the user to understand the structure of the generated code.

1.4.4 Minimal Invasiveness

To achieve all of these goals, it is necessary to modify the code generation process. However, an important goal is to ensure that these modifications are as minimally invasive as possible. One important reason is that compilers, particularly compiler optimizations, are correctness-preserving, which means that a correct source program will be translated to a correct target program (if no bugs in the compiler are present). Copperman [Cop94] emphasizes that this means a correct compiler may still change the behavior of incorrect programs through optimizations, perhaps even making them correct. Therefore, the same code that is debugged should be deployed later on to ensure that all bugs have been found. Introducing a performance overhead to facilitate debugging will thus also slow down the finished product.

Another reason is to ensure simplicity of the generated code to allow for easier verification and validation. Section 2.4 gives more details on formal verification and why simple code is preferrable for that purpose.

1.5 Outline

The next chapter highlights some existing work from a wide range of topics related to this thesis. Debugging in different variants, formal verification as an alternative to debugging generated code, algorithm visualization as a form of visual feedback on the program's execution and other topics are covered. Chapter 3 explains technologies used in the implementation of the demonstrator. In Chapter 4, design considerations, breakpoint semantics, visual syntax and other preliminary decisions are discussed before Chapter 5 gives details on how the decisions made before were realized in the concrete implementation. An evaluation of the implemented demonstrator is presented in Chapter 6, before Chapter 7 concludes with a summary of what has been achieved and an outlook at future work.

Chapter 2

Related Work

This chapter presents a selection of publications thematically related to this thesis in various ways. Each publication is summarized and put into perspective regarding my topic.

In Section 2.1, the family of synchronous languages and the SCCharts language are explained since both some related work and the concepts presented later require a basic understanding of the synchronous model of computation. Section 2.2 presents different approaches to compiling model-based languages. Lena Grimm's Bachelor's thesis on debugging SCCharts is presented in Section 2.3. Afterwards, Section 2.4 deals with formally verified compilers since they offer an alternative to manually debugging and verifying generated code.

Section 2.5 presents existing applications of compiler tracing in debugging while Section 2.6 explains different approaches to visualizing the runtime state of a program. Finally, other approaches to easier debugging of model-based code with semi-automatically generated debuggers in Section 2.7 and a model-level debugger for cyber-physical systems in Section 2.8 are shown.

2.1 Synchronous Languages

Synchronous Languages are a set of programming languages mainly used for safety-critical real-time systems. Many synchronous languages follow a model-based approach and are compiled to a host language such as C or Java. The generated code can then be further compiled using a standard compiler for that language. For this reason, the debugging concepts presented in this thesis can be applied to them well. Even though each one has its own *Model of Computation (MoC)*, most of them are based on the *Perfect Synchrony* hypothesis [BG92]:

A perfectly synchronous system executes each one of its reactions in zero time. Therefore, all outputs are generated at the same time the corresponding inputs are read. The execution is divided into logical *ticks*, where each tick is conceptually instantaneous and consists of reading inputs and producing the corresponding outputs. Each *tick* or *macrostep* consists of a finite number of *microsteps*, each of which is executed atomically and synchronously. This means that on a real machine, each tick can be executed in a finite amount of time, while a program run can be infinite since it may contain an infinite number of ticks.

With synchronous languages, deterministic concurrency can be achieved [BCE+03], and since their semantics are often mathematically defined, programs written in them are generally well-suited to be formally verified.

2.1.1 Constructiveness and Signals

To achieve deterministic concurrency, perfect synchrony does not suffice. Defining all reactions as instantaneous may still lead to race conditions if different threads write to the same variable in the same tick. Therefore, additional semantic restrictions need to be defined and checked at compile time. To this end, many synchronous languages use *signals* and check the program for *constructiveness*.

A signal is similar to a physical wire in a circuit and can either be `present` or `absent` in each tick. The state of the signal is not preserved across tick boundaries. It is considered `present` if and only if it is `emitted` during the current tick and will remain so throughout the entire (conceptually instantaneous) duration of it. One of the first languages to introduce the concept of perfect synchrony and signals was *Esterel* [Ber00], a textual language developed by Gérard Berry and his team.

```
Present X else emit X;
```

Listing (2.1) A contradictory Esterel program. The signal changes state after it was already read in the same tick, making its state non-consistent.

```
Present X then emit X else emit X;
```

Listing (2.2) A logically correct, but non-constructive Esterel program.

Figure 2.1. Example Esterel Programs.

When executing an Esterel program, each signal must be assigned a unique value for each tick. If there is no such assignment or multiple consistent options exist, the program is *logically incorrect*. Listing 2.1 shows such a program. However, even if there is exactly one valid assignment, that assignment may be hard to find.

Listing 2.2 shows a program which is logically correct. `X` has to be `present` since otherwise, the `else`-branch would lead to `X` becoming `present` later after being read as `absent`, which is not allowed. However, there is no way to determine the value of `X` without speculatively executing all possible program paths.

A *constructive* program must have a unique valid assignment for each signal in each tick that can be computed without speculative execution. Constructiveness is thus a restriction of logical correctness. For example, adding `emit X;` as the first line in Listing 2.2 would mean that without any speculative execution, the compiler can determine that `X` *must* be present. This way, an assignment is found and can be validated without needing to try every possible value for `X`. With the added `emit` statement, Listing 2.2 becomes constructive; without it, it is rejected by the compiler.

2.1.2 SCCharts and the Sequentially Constructive MoC

For state-based contexts, e. g., in automation or safety, formalisms based on *Mealy automata* [Mea55] are commonly used. David Harel [Har87] proposed *Statecharts*, an extension of these automata with hierarchical and parallel components. Charles André developed a synchronous implementation of Statecharts called *SyncCharts* [And95].

Sequential Constructiveness

With perfect synchrony and constructiveness checks, many synchronous languages such as SyncCharts offer determinism and sound semantics, leading to safer programs that are easier to verify than non-synchronous ones [BBD+17]. However, the compiler checks greatly restrict the set of programs that are accepted, imposing a steep learning curve and reduced flexibility on the programmer, especially when accustomed to non-synchronous imperative languages.

Therefore, von Hanxleden et al. [HMA+14] proposed a conservative extension of the perfectly synchronous MoC of SyncCharts called the *Sequentially Constructive (SC)* MoC. This extension relaxes the constructiveness requirement to make programming easier while not sacrificing any soundness of the associated semantics.

In the SC MoC, all variable and signal operations in concurrent regions are scheduled according to the *Initialize-Update-Read (IUR)* protocol. This means that variable initializations (assignments) are always scheduled before updates (e. g., increments), which in turn come before read operations. Multiple parallel operations that are associative (e. g., parallel increments) are allowed since they can be scheduled arbitrarily without influencing the outcome. Conflicting ones where the IUR protocol does not define a clear order (e. g., parallel initializations with different values) are detected and rejected by the compiler. Sequential non-concurrent accesses can manipulate a variable an arbitrary number of times even within the same tick since this cannot introduce any ambiguity.

With this extension, the programmer is more flexible in writing their code without being restricted as heavily as with a perfectly synchronous MoC. All sequentially constructive programs can be statically assigned a scheduling for their variable accesses, which is then applied during code generation to ensure a deterministic program outcome.

SCCharts and their Semantics

An extension of SyncCharts based on the SC MoC is *SCCharts*, proposed by von Hanxleden et al. [HDM+14]. Much like SyncCharts, SCCharts is a graphical state-machine based language designed for programming safety-critical real-time systems and is used today in industrial contexts such as the development of railway interlocking systems.

Figure 2.2 shows an example SCChart called `ABRO`. It waits for two boolean inputs A and B before it sets its output O to `true`. Whenever R is present, the program and the output are reset.

SCCharts are inherently concurrent; parallel regions such as `HandleA` and `HandleB` are executed concurrently. Dashed transitions are *immediate*, meaning they can be executed in the same tick their source state is entered as long as their trigger evaluates to `true`. Other transitions only become active in the next tick after their source state has been entered, so for example, the ABRO SCChart reacts to neither A, B nor R in the first tick of its execution since none of the respective transitions are active yet.

To leave hierarchical states with inner behavior, three types of transitions can be used. A *termination transition* is indicated by a green triangle at the start and can only be taken after all of the source state's inner regions have reached a final state, which are indicated by a double border. `Weak abort` transitions are plain arrows without any additional markers and can be taken any time their trigger is `true`, provided they are immediate or the state has been entered in a previous tick. When a state is left through a weak abort transition, the behavior of all of its inner regions is executed for the rest of the tick before being aborted. Contrarily, a `strong abort` transition, indicated by a red circle at its start, will immediately exit the state when triggered, without executing any of the inner behavior.

Each state may have `entry`, `exit` and `during` actions. These are executed when the state is entered or left or once per tick in which the state is active, respectively. For example, the `entry` action on state `ABO` will cause the output `O` to be set to `false` in the first tick and after each reset.

Various other complex features are available, however, they are not relevant here and thus omitted.

2.2 Model-Based Compilation Approaches

To compile SCCharts and most other model-based languages, multiple steps are used. The model is often passed through a series of m2m transformations, replacing complex features such as aborts with multiple semantically simpler constructs, before the actual gode generation step. While this process makes further compilation simpler since fewer constructs need to be considered, the model structure may be heavily altered, making the transformed model and consequently the generated code hard to understand for the user.

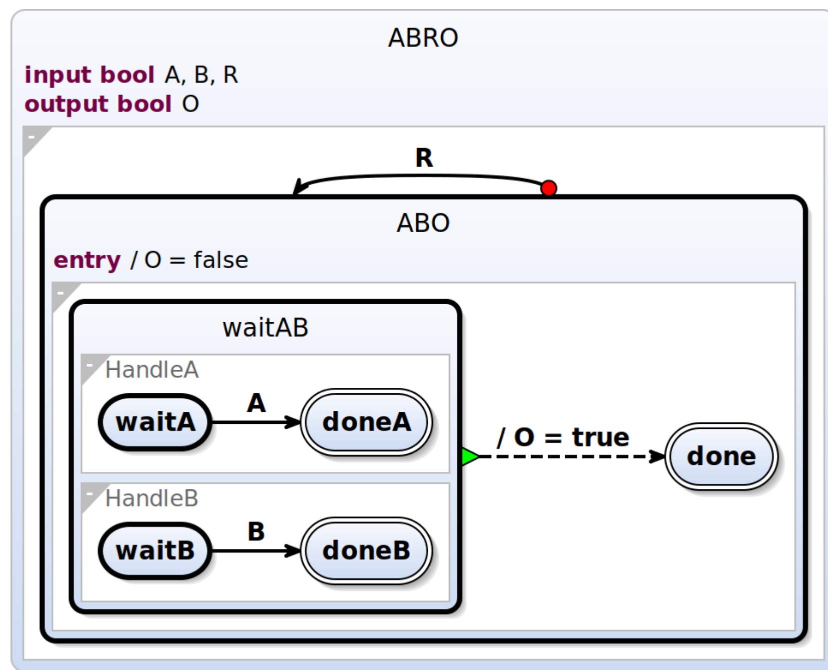


Figure 2.2. An Example SCChart.

Three different approaches to compiling model-based languages are presented here, all of which are available for SCCharts. Chapter 4 discusses ways in which the debugging concepts proposed in this thesis can be applied to all three. However, the demonstrator has been implemented exclusively for the state-based approach since it is best suited for model-based debugging.

2.2.1 State-Based Compilation

The state-based compilation approach was proposed by Smyth et al. [SMH18], and contrary to earlier proposals [HDM+13], it focuses more on code readability than on pure performance. The design idea is to generate code that would be easy to relate to the source code in order to facilitate verification and general understanding. This approach is therefore likely to be used in scenarios where debugging and verification are desired and is thus particularly suited for demonstrating the debugging concepts proposed in this thesis.

All SCCharts compilation systems start by transforming complex features away, yielding so-called *core SCCharts* with a reduced set of features. The state-based code generator then generates a Java class consisting of the following components:

- ▻ an `Interface` object containing all input and output variables of the SCChart. All fields of this object are directly writeable.
- ▻ a `TickData` object containing contexts for all root-level regions.
- ▻ one enum per region containing all states of that region.
- ▻ one `Context` class for each region. This class contains information on the region's thread status (whether the region still has work to do in the current tick) as well as the active state of that region

and contexts for subregions.

- one method per state. This method calls all of its subregions' methods in scheduling order, then checks whether a transition needs to be taken. Transitions are checked in priority order, and if one transition's guard evaluates to `true`, the context's active state is set to the transition's target state and the control is passed back to the parent region of the current state. An example method generated for a state can be seen in Listing 2.3.
- one method per region, such as the one shown in Listing 2.4. This method simply calls the appropriate state method for the current active state of that region found in the corresponding `context` object.
- `reset` and `tick` functions to be called from outside the generated code

```
1  private void ABRO_statewaitA(ABRO_regionR2Context
       context) {
2    if (context.delayedEnabled && (iface.R)) {
3      context.delayedEnabled = false;
4      context.activeState = ABRO_regionR2States.DONEA1;
5    }
6    else if (context.delayedEnabled && (iface.A)) {
7      context.delayedEnabled = false;
8      context.activeState = ABRO_regionR2States.DONEA1;
9    } else {
10     context.threadStatus = ThreadStatus.READY;
11   }
12 }
```

Listing (2.3) Code generated for a state

```
1  private void ABRO_regionR1(ABRO_regionR1Context
       context) {
2    while (context.threadStatus == ThreadStatus.
       RUNNING) {
3      switch (context.activeState) {
4      case WAITAB:
5      ABRO_statewaitAB(context);
6      // Superstate: intended fall-through
7
8      case WAITABRUNNING:
9        ABRO_statewaitAB_running(context);
10       break;
11
12     case DONE1:
13       ABRO_statedone1(context);
14       break;
15
16     case _AC1:
17       ABRO_state_AC1(context);
18       break;
19     }
20   }
21 }
```

Listing (2.4) Code generated for a region

Figure 2.3. Code extracts generated with the state-based approach. Names shortened for better readability.

2.2.2 Priority-Based Compilation

The priority-based compilation approach was originally proposed by von Hanxleden et al. in 2014 [HMA+14] and extended by Lars Peiler in 2017 [Pei17]. As opposed to the state-based approach, no static schedule is computed. Instead, lightweight application-level threads and a dynamic scheduler are used. This approach brings faster average execution times than both the netlist- and state-based approaches [Pei17; SMH18] as well as a drastically reduced number of generated variables compared to the netlist-based approach, which is particularly important for embedded applications with low runtime memory such as the Lego Mindstorms Platform [SMS+19]. Another advantage over the original netlist-based approach is that the priority-based compilation supports certain types of immediate loops, which are rejected by the netlist-based compiler.

Normalization and SCG

As with all SCCharts compilation chains, the first step in priority-based compilation is to transform the model until it only contains a small set of constructs that are straightforward to compile further. This transformation ends when a *normalized SCChart* has been created. Normalized SCCharts contain an even further reduced set of features than core SCCharts and only consist of the following features:

- immediate transitions with a single effect and no trigger
- immediate transitions with a trigger and no effects
- delayed transitions with neither trigger nor effects
- parallel regions and hierarchy
- final states and termination transitions

This normalized SCChart is then mapped to a *Sequentially Constructive Graph (SCG)* consisting of statement nodes containing variable assignments, fork / join constructs or conditional expressions as well as control flow edges that can be either delayed or immediate. On this SCG, the analysis of dependencies and the assignment of priorities can be performed.

Dependencies and Priority Assignment

As mentioned in Section 2.1.2, SCCharts follow the *Initialize-Update-Read (IUR)* protocol. When the SCG has been generated, IUR dependencies are added as extra edges. For example, an initialization of a variable has to be scheduled before a concurrent read access to the same variable, therefore an edge starting at the initialization and ending at the read is added.

In the next step, each node is assigned a *node priority*. Nodes with a higher priority must be executed before concurrent ones with a lower priority; if two nodes have the same priority, their order of execution does not matter.

Since the IUR protocol only affects actions within the same tick, delayed edges do not need to be considered. Surface nodes, located at the start of each delayed edge, as well as the program's exit node are assigned priority 0. From then on, each node without outgoing IUR edges is assigned the maximum priority of its successors. Nodes with outgoing IUR edges receive either their sucessors' priority or the dependency edge's target node's priority plus one, whichever is higher.

This way, it can be ensured that the source node of the dependency always receives a higher priority than that of the target node and the priorities in sequential statements monotonically decrease.

From these priorities, each thread is assigned a *priority identifier (prioID)* based on its thread ID and the node priorities it contains. Nodes with lower priorities generally receive lower prioIDs. While non-concurrent threads may have the same prioID, statically concurrent threads must not. Therefore, the prioIDs can be used to uniquely identify each thread.

An example of an SCG with assigned node priorities and prioIDs can be found in Figure 2.4.

Runtime Scheduling

While the original priority-based compilation approach uses C as a target language and heavily relies on preprocessor macros for scheduling, a Java version exists that uses a standard superclass containing all required methods as a basis for priority-based code. For consistency among the approaches, the Java version will be considered here.

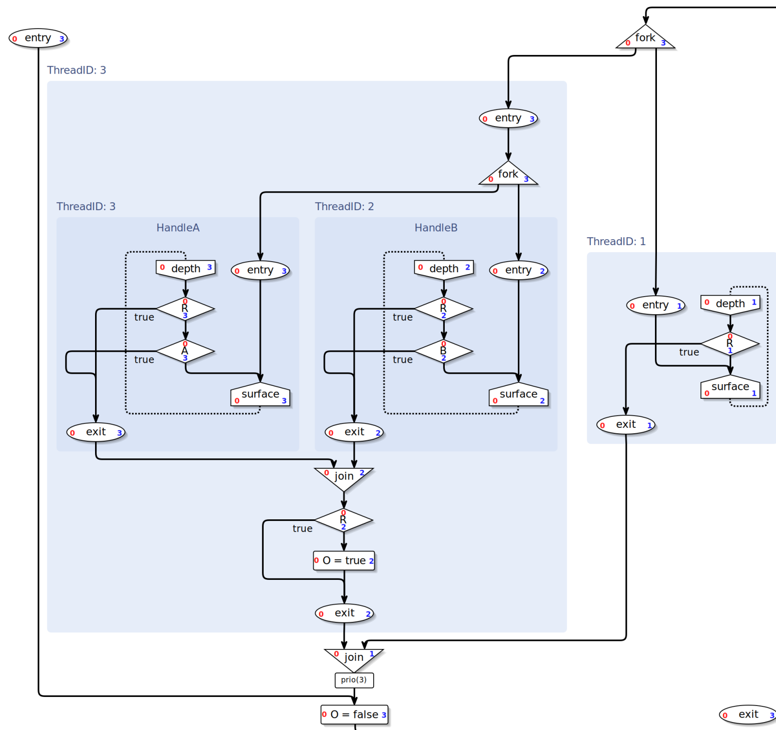


Figure 2.4. SCG of the ABRO example with assigned priorities. Node priorities are red while prioIDs are blue.

In code generation, an enum containing labels for various code locations is created. These include entry labels for each region of the SCChart as well as labels for each place where a priority change happens, i. e., where an SCG node with an outgoing IUR edge was located.

The current state of each application-level thread is kept using various arrays. These keep track of active threads (threads that still have work to do in the current tick), alive threads (threads that have been forked, but not yet joined) and for each thread, the label where its execution is to be resumed, i. e., a program counter.

While there are still active threads, the one with the highest priority is set as active thread and a portion of its code is executed. The program logic itself is organized in a large `switch` statement that executes a section of code based on the current program counter of the active thread. An extract of such a `switch` statement can be seen in Listing 2.5. The section ends when the program reaches either a join, a delayed edge (in which case the thread becomes inactive), or a priority change (in which case the thread remains active, but yields control).

Whenever control is taken away from a thread, its program counter is updated to the label where it stopped and the active thread with the highest priority is scheduled again. This process repeats until there are no active threads left, at which point the tick ends. At the beginning of the next tick, all alive threads become active again.

```
while (!isTickDone()) {
  switch (state()) {
    case ABROEntry:
      O = false;
      fork(State._L_1, 1);
      gotoB(State._L_0);
      if (true) break;

    case _L_0:
      fork(State.HandleB, 2);
      gotoB(State.HandleA);
      if (true) break;

    case HandleA:
      pauseB(State._L_2);
      if (true) break;

    case _L_2:
      if(R){
        gotoB(State._L_3);
      } else {
        gotoB(State._L_4);
      }
      if (true) break;

    case _L_3:
      termB();
      if (true) break;
    [...]
  }
}
```

Listing (2.5) Code extract from priority-based code

```
public void logic() {
  _g5 = _pg14_e1;
  _g10 = _pg14;
  _g14_e1 = !(_g5 || _g10);
  _g6_e1 = !_g5;
  _g7 = _g5 && !R;
  _g5 = _g5 && R || _g7 && A;
  _g11_e2 = !_g10;
  _g12 = _g10 && !R;
  _g10 = _g10 && R || _g12 && B;
  _g11_e2 = (_g6_e1 || _g5) && (_g11_e2 || _g10) &&
      (_g5 || _g10);
  _g11 = _g11_e2 && !R;
  if (_g11) {
    O = true;
  }
  _g6_e1 = _g11_e2 && R || _g11;
  _g6 = _pg20;
  _g15 = !_g6;
  _g13 = _g6 && R;
  _g19_e2 = (_g14_e1 || _g6_e1) && (_g15 || _g13) &&
       (_g6_e1 || _g13);
  _g19 = _GO || _g19_e2;
  if (_g19) {
    O = false;
  }
  _g14_e1 = _g19 || _g7 && !A;
  _g14 = _g19 || _g12 && !B;
  _g20 = _g19 || _g6 && !R;
}
```

Listing (2.6) Code extract from netlist-based code

Figure 2.5. Example code generated with priority- and netlist-based approaches.

2.2.3 Netlist-Based Compilation

Similarly to the priority-based approach, the source model is simplified through a set of m2m transformations until a *normalized* SCChart with a reduced set of features is left. This model is then mapped to an SCG. However, since this approach does not use dynamic scheduling, the concurrency within the SCG needs to be eliminated and the SCG must be sequentialized.

To this end, the SCG is split into *basic blocks*. A basic block is a set of SCG nodes that will always be executed in the exact same order, i. e., it does not contain any conditionals or incoming dependency edges. The former may lead to different executions depending on the conditional's result while the latter may mean that between the node with the incoming edge and its predecessor, a node in another thread must be executed to satisfy the dependency. Delayed edges also separate basic blocks.

For each basic block, a *guard* is computed. The guard is a boolean value that determines whether the block should be executed. For the first block of the program, the guard is `true` in the first tick. For blocks following after a conditional statement, the guard is `true` iff the conditional's guard is `true` (i. e., the conditional is executed this tick) and the conditional's result corresponds to the block's branch condition. If a block follows after a delayed edge, its guard is `true` if the guard corresponding to the block before the delayed edge was `true` in the previous tick. Blocks that start with an incoming IUR dependency have their guard as `true` if the guards of both the previous block in the same thread and the origin of the IUR edge are `true`.

The guard's computations are added as assignment nodes to the SCG, located at the beginning of their respective basic blocks. The blocks are then sequentially ordered in a way that the sequential control flow within each thread is preserved while all concurrent variable accesses (including the guards) follow the IUR protocol. Delayed edges can be removed since they are now represented by the guards being stored across ticks. If such an ordering cannot be found, e. g., due to circular dependencies between parallel threads, the program is rejected.

Since a logical circuit is to be synthesized and variables should behave similar to wires in a real circuit, they cannot change values arbitrarily. To achieve a stable circuit, wires should retain their values throughout a tick, similar to signals in perfectly synchronous languages as presented in Section 2.1. In SCCharts, the SC MoC allows for variables and signals to change values during a tick, however. Therefore, multiple assignments to the same variable are split into separate instances, similar to *Static Single Assignment (SSA)* [App98]. This is only possible if there is a finite (and statically known) number of assignments to each variable in a tick, thus the netlist-based approach conservatively rejects models containing any immediate control flow cycles.

Once sequentialized and split, an SCG with no concurrency, single assignments to each variable and no delayed edges is left. This SCG can then be translated to a circuit network comprised of gates and registers that can be represented by either a program or a real circuit. A Java example of such a circuit network can be seen in Listing 2.6.

2.3 Debugging SCCharts

One closely related work is Lena Grimm's Bachelor's Thesis on debugging SCCharts [Gri16]. Grimm proposed a debugger for SCCharts that allows the user to place breakpoints in SCCharts models, which then suspend the model's execution in the appropriate place when run in a simulator. Breakpoints are placed in the model's source code using the text editor provided for SCCharts editing much like line breakpoints in any general-purpose language. The breakpoints are then also visually indicated in the automatically generated diagram. The functionality of the tool is highlighted in Figure 2.6.

Since the debugger is coupled to a simulation on model level, it is mainly useful for debugging SCCharts models on their own and for finding bugs on the model level. The approach is completely independent from the compiler used for code generation. It was integrated seamlessly into the SCCharts development workflow and worked much like what developers are used to from other languages.

One drawback resulting from the use of the simulator is that some breakpoints cannot suspend the execution immediately since the simulator only supports a macro-level step size. For that reason, the execution sometimes suspends in a location with no breakpoint because the execution has passed through a breakpoint in the previous step, leading to possible confusion.

The main difference to this thesis is that due to the isolated nature of the simulation, no programmatical interaction with the model is possible. For example, having multiple models ticking one after the other and possibly influencing each other's interface variables or having host-code calls from within the model is not possible. Therefore, the model itself can be debugged well, however, integration tests of the entire system consisting of generated code from potentially multiple SCCharts models, wrapper code handling I/O for the models and other host code interacting with the models are not supported.

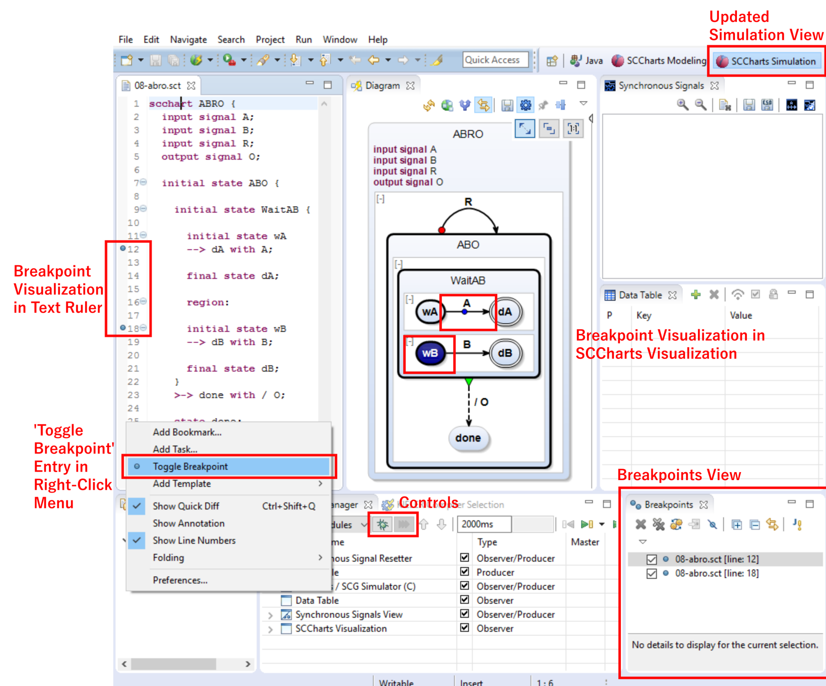


Figure 2.6. SCCharts debugging components added by Lena Grimm [Gri16]

2.4 Formal Verification

One key reason why debugging the generated code rather than the source model is essential is the fact that most compilers, even though thoroughly tested, have never been formally verified. Therefore, one cannot simply assume that the generated code is semantically equivalent to the source model. Particularly in safety-critical applications, even small uncertainties are not tolerable.

If one wants to ensure correct behavior of the generated code, only two options remain. One option is to prove the desired properties for the generated code itself, e. g., using *model checking*. Model checking is a technique where properties of programs can be modelled and checked as logical formulae, e. g., in *Linear Temporal Logics (LTL)*. Tools for model checking are available for most high-level languages, such as the one for SCCharts proposed by Andreas Stange [Sta19]. With this method, errors introduced by the compiler can be caught since the generated code is examined directly. However, formulating and testing sufficient properties becomes more and more difficult with increasing code complexity and decreasing abstraction level. Therefore, proving correctness properties on generated code is generally undesirable.

The second option is to use a formally verified compiler similar to the one Bourke et al. [BBD+17] propose for Lustre, a synchronous dataflow language used in many safety-critical infrastructure applications. For their compiler, they formally specify the semantics of source, intermediate and target language and prove that any source program is mapped to a fully equivalent target program.

Even though formally verifying an entire compiler is a complex and time-consuming task, it is then sufficient to only prove correctness properties on (relatively simple) source models with existing model checkers and other tools for high-level languages, since the semantic equivalence proof of the compiler

guarantees that said properties will also hold for the generated code. However, using formally verified compilers is only suitable for languages whose syntax and semantics are fixed, since any change to either will lead to the need for a re-verification of the compiler. It is important to note here that the target program generated by the Lustre compiler is not yet executable, but rather a C program that needs another compilation step using a regular C compiler. To ensure that the semantic equivalence is preserved through this step as well, Bourke et al. use a formally verified compiler for a subset of C called CompCert [Ler09].

2.5 Compiler Tracing and its Applications

A key compiler feature used for this thesis is the *tracing* functionality, i. e., a mapping between source and target model elements that enables the user to track model elements and their derivatives through the compilation process. With that knowledge, a model-based debugger can relate generated code to the source model elements it was created from, allowing the debugger to both find appropriate locations in the generated code to set breakpoints for the user and to extract runtime information from the generated program during debugging. Section 4.2 describes how tracing information can be used for these purposes.

Exploiting tracing for debugging is not a new concept, however. This section presents a set of debugging applications where the same technology is used.

2.5.1 Performance Debugging

In their 2008 paper [JHR+08], Ju et al. propose a new approach to *Worst-case Execution Time (WCET)* analysis for Esterel code using compiler tracing functionality.

As stated in Section 2.1, in synchronous languages such as Esterel, it is assumed that each tick is performed instantaneously with no time passing in between microsteps. However, since in reality, such programs are compiled to a host language (e. g., C), microsteps are in fact executed sequentially and there may well be time passing in between two microsteps, for example due to scheduling interference. However, a generated host language program still effectively meets the requirement of perfect synchrony if a tick is always completely computed before the next set of input events arrives, guaranteeing that inputs remain constant throughout a tick. The WCET analysis therefore computes an upper bound to the maximum time a tick's computation may take to ensure that the synchrony assumption is never violated.

Due to the fact that the code under analysis is always generated from Esterel models, Ju et al. are able to find tighter estimates for the WCET in their particular case than other, more generic tools can. Additionally, they employ advanced infeasible code analysis and allow the user to place source-code level annotations to signal to the tool that certain paths cannot occur in a real program run.

However, their main contribution is a critical path highlighting in the source model. Whenever the WCET analysis finds a possible execution that takes longer than the time between inputs, it would usually be hard for the programmer to determine what parts of the source model contribute to the path being too slow, especially for complex models. The tool proposed by Ju et al., however, automatically highlights all source statements contributing to the critical path, making it immediately clear to the programmer where the problem is caused. This critical path highlighting on model level can be seen in Figure 2.7.

Contrary to the concepts proposed in this thesis, this approach does not directly contribute to debugging a program in the classical sense. However, it still helps the user improve their generated code by pointing out potential problems in the source model, thus following a very similar approach.

```
1   module reflex_game
    …
7   relation ..., READY # STOP
    …
14  every COIN do
    …
22    [
23        copymodule AVERAGE
24             ||
          …
38             trap END_MEASURE in
39             [
40                every READY do
41                   emit RING_BELL
42                end
43                ||
         …
52       do
53          do
54             every MS do
55                   TIME:=TIME+1
56             end
57             upto STOP;
58             emit DISPLAY;
59             emit INC_AVE(TIME)
60          watching LIMIT_TIME MS
61          time out exit ERROR end;
62          emit GO_OFF;
63          exit END_MEASURE
64       ]  %trap END_MEASURE
             ...

87  module AVERAGE
    …
91     every immediate INC_AVE do
92        TOTAL := TOTAL + ?INC_AVE
93        NUM := NUM +1;
94        emit AVE_VALUE (TOTAL/NUM)
95     end
    …
```

Figure 2.7. Example Esterel program with critical path highlighting as proposed by Ju et al. [JHR+08]

Also, even though slow execution is not primarily seen as a bug by many programmers, it may cause behavior not matching the specifications in safety-critical real-time applications and therefore needs to be eliminated just as any other bug would.

2.5.2 Debugging Optimized Code

Another area where it is necessary to find an appropriate mapping between written and actually executing code is in debugging in the presence of compiler optimizations. In 1994, Copperman [Cop94] described the problem of optimizations in combination with debugging. He points out that even though compiler optimizations should be *correctness-preserving*, they may still change the behavior of code being debugged since that code is unlikely to be correct. Therefore, it is necessary to debug the code with the optimizations, which may in turn lead to issues with breakpoint locations and variable values.

In non-optimized code, there often is a block of machine instructions associated with each source statement. If that is the case and the blocks are concatenated in the order of statements in the source model, placing a breakpoint at the start of that block will implement the desired breakpoint semantics. With optimizations in place, however, there may not be an obvious place to put a breakpoint since machine instructions may be left out or swapped around for faster execution, interleaving parts of consecutive source statements and thus removing clear statement boundaries.

Another issue described by Copperman is caused by *non-current variables*, i. e., variables that do not have the value one would expect from reading the source code. For example, constant propagation and dead code elimination may lead to variable assignments being deleted. If variable `a` is assigned a constant value `c` and then variable `b` is assigned `a`, one would expect that `a = c` after the last statement. However, a compiler with optimizations enabled may delete the assignment to `a` if the value of `a` is never read again and directly assign `b = c`. A debugger queried for the value of `a` would then display an unexpected, or *non-current* value for a.

Copperman therefore presents a method to automatically determine whether variable values in optimized code are non-current and if so, to display a warning to the user along with an explanation of how the value came to differ from the expected one. To achieve this, flow graphs storing the relation between declarations and memory locations are used. To map between the non-optimized and optimized versions of these graphs, compiler tracing is used. Analyses on these data structures can be conducted to find parts of statements that have been moved by optimizations as well as determine non-current variables and the cause of their unexpected values.

Copperman follows a similar goal to that of this thesis, namely to close the user's mental gap

between generated and original code. The alterations introduced by compiler optimizations may be similar to those made by m2m transformations in model-based design. However, Copperman's main focus lies on individual variables and their values while this thesis aims to visualize the program's memory state as a whole.

2.5.3 Transparent Debugging of Dynamically Optimized Code

Traditional compiler optimizations are applied statically at compile time, resulting in binary code that does not necessarily resemble the source program's structure. In contrast to that, *dynamic optimization* occurs at runtime and is achieved by analyzing the runtime behavior of the program before adding optimizations to it. A key challenge in this field is the instrumentation code introduced at compile time to switch control between the actual program and the optimizer periodically. This serves to regularly permit the optimizer to perform optimization passes, altering the code structure according to an analysis of the past program performance. Not only does the debugger need to find an appropriate mapping between source code and dynamically changing optimized code, it also needs to filter out all actions performed by the instrumentation code and the optimizer to hide them from the user.

Kumar et al. [KCS09] propose a debugging framework called *DeDoc* to transparently debug code even in the presence of dynamic optimizations. To meet the requirements listed above, they use a virtual debugging environment in which the program under debugging runs. The program is divided into *traces*, i. e., blocks of instructions without jump instructions, and control is transferred to the optimization engine after each trace. The engine can then either run an optimization pass on the previous trace if that trace has been executed sufficiently often that optimizing it may be worthwhile, or dispatch the next trace.

Native debuggers can then be adapted to not target a regular execution of the program under analysis, but rather the debug engine. The engine will keep track of all dynamic optimizations and generate debug information for the native debugger to mask the fact that the optimizations have occurred dynamically, effectively allowing the debugger to apply the same methods as for statically optimized code, which most modern compilers can handle already.

Especially for model-based languages with dynamic optimizations or models intended for running on systems with native dynamic optimizers, the concepts presented by Kumar et al. may need to be considered for model-based debugging, too.

2.6 Visual Debugging

A key goal of this thesis is to provide visual feedback on a running program, effectively visualizing the algorithm modelled by it. However, the concept of algorithm visualization is not new.

In their 1994 paper on visual debugging [MS94], Mukherjea and Stasko propose a concept to easily create algorithm animations for debugging. According to them, *algorithm animation* is a technique to visually present the "big picture" of an algorithm, allowing the viewer to understand the fundamental concept behind it. Since those animations are very specific to the algorithm that is being visualized, they are hard to automatically generate and are usually made by hand for teaching and demonstration purposes.

Contrarily, *data structure visualization* automatically synthesizes views of low-level data structures at runtime, which may be useful for debugging. However, such visualizations cannot convey the full concept of the program and are only useful if the idea behind the program has already been understood by the user.

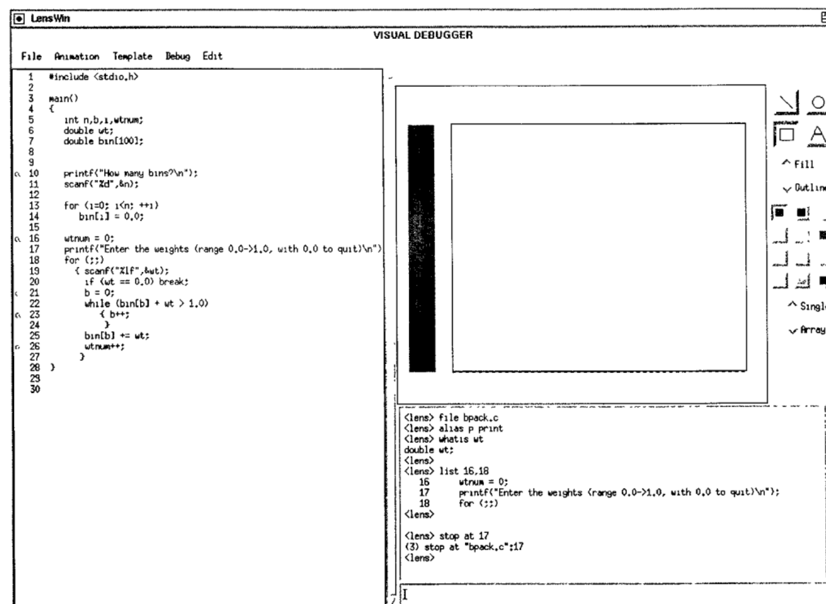


Figure 2.8. Visual debugging tool *Lens* as proposed by Mukherjea and Stasko [MS94].

The approach of Mukherjea and Stasko is to provide a simple tool for algorithm visualization that allows programmers to quickly make a visualization of their system under development without needing extensive knowledge of graphics programming or the specific animation toolkit to allow the use of animated views even during debugging.

To achieve this, they built a tool called *Lens* that incorporates a palette-based graphical editor to create animations, a source code view and a debugger console in the same window. The tool can be seen in Figure 2.8. The available animation components have been selected based on a study on animations created with previous tools and the most commonly used features there. The debugger can be used to step through the program at runtime while the animation window will display the current state of the program according to the model built by the programmer.

Mukherjea and Stasko come to the conclusion that their system is well-suited for rapid prototyping of animations and the visualization of small, "classical" computer science problems such as sorting or graph algorithms. However, the ease of use comes at the cost of a reduced set of features that leads to less flexibility when designing more complex animations.

2.7 Semi-Automatic Debugger Generation

When implementing *Domain-Specific Languages (DSLs)* manually, a lot of components are either necessary for the language's execution (e. g., parsers, interpreters / compilers) or desirable for an efficient workflow (e. g., IDE support such as syntax highlighting, error markers, outline views). While implementing all these components by hand can be time-consuming and repetitive, modern *language workbenches* such as *Xtext* [EV06] can automatically generate many of them from language specifications.

Lindeman et al. [LKV11] propose an extension to such a language workbench that also generates debuggers for DSLs with minimal effort. Usually, specifying a debugger and correctly integrating

it into an existing IDE is tedious and complex since many components of the debugger (such as debug events, a custom debug model and a matching frontend) need to be integrated. Also, debuggers are specific to the language they were written for, since they need to navigate the language's scope hierarchy. When a session is suspended, debuggers must display meaningful information relating the current execution state with the source code. This means that for every DSL, a separate debugger needs to be implemented.

Lindeman et al. propose a modular debugger system with a generic debugger backend that is integrated into the Eclipse IDE as well as a declarative system to specify debuggers for DSLs that can communicate with the backend. Their system is based on aspect-oriented programming concepts where a user declaratively specifies patterns for code locations where debug events would need to occur (e. g., where a program step ends) along with DSL code snippets that emit those events. The system then automatically adapts every program during compilation before it is translated to the host language to ensure proper communication with the debugger. Based on the different types of events specified for the language, a debugger module is generated on top of the generic backend.

This approach guarantees that when debug events are generated, the original model structure is available and therefore, mapping the events to source code locations is simple. Also, the fact that only DSL constructs are added makes the specification independent of the compilation process and can thus support multiple compiler backends with ease.

The resulting debuggers allow the execution of generated code from a DSL while stepping through the source model in a user-defined granularity. Using this concept, debuggers implemented for model-based debugging could be extended to support stepping not only on the host-language level, but also on the model level. For SCCharts, this could mean stepping over a tick or skipping the execution of a certain region independently from the code generation approach. Due to the semi-automatic generation, the effort for this method would be lower than manually implementing such functionality.

2.8 Model-Level Debugging

In the domain of *Cyber-Physical System (CPS)* programming, software is written to control physical entities using sensors and actuators. While these systems have high requirements in terms of safety, they are also often programmed using a variety of different languages.

Djukić et al. [DPL16] describe an example scenario using three different types of model-based DSLs to program a single robot arm: one for control logic, one for specifying physical properties of the robot arm and one that describes the environment where the system will operate. All of these languages are compiled to a common host language and executed together, which makes it difficult for the programmer to understand what parts of the generated code originate from what modelling language and how they interact with one another.

To tackle this problem, Djukić et al. propose an interactive debugging environment where the source models are automatically compiled each time they change and a dynamic visualization of the entire source system is generated. An example visualization of the robot arm system can be seen in Figure 2.9. This visualization shows a diagram of the different model parts written in different languages, circuit-like connections where values are exchanged and sliders and switches to allow the user to dynamically set model parameters or sensor inputs. The system can then be either simulated locally or executed on multiple real hardware devices connected to the debugging system to allow for a realistic debugging scenario.

Djukić et al. focus on the interactions between different submodels developed in different languages and visualizing the system as a whole in a novel way. The integrated simulation capabilities and the integration of debugging on the target devices shows the focus on low-level control systems. In

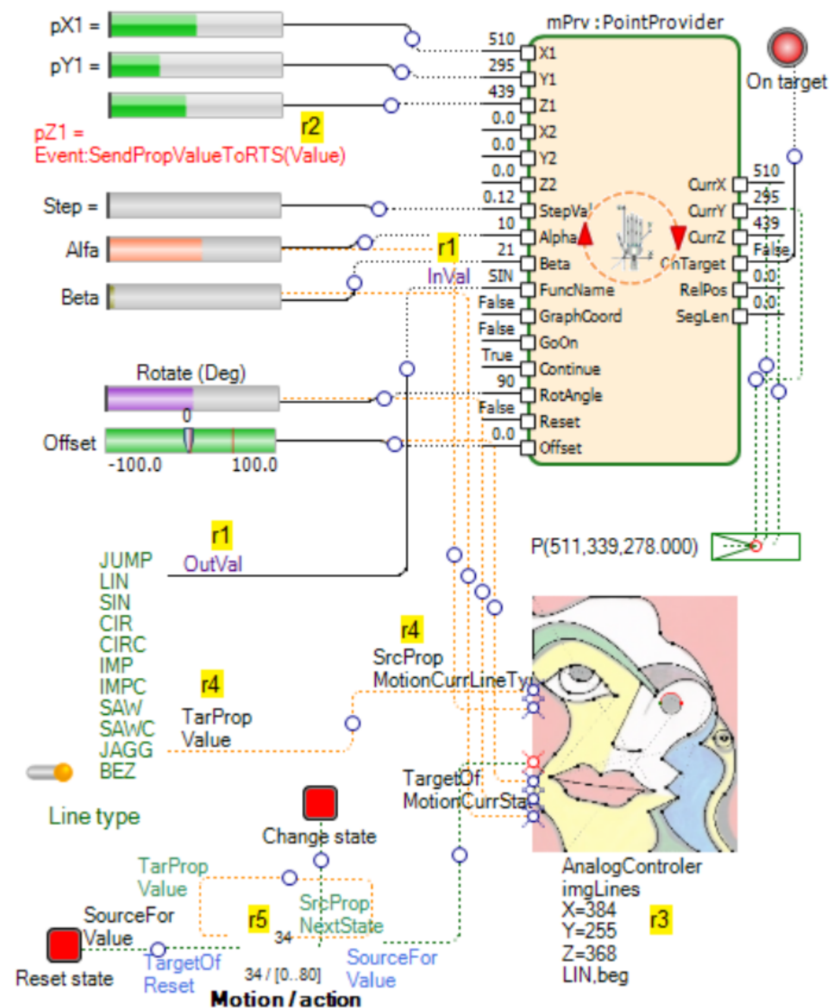


Figure 2.9. An example system as visualized by the model-level debugger proposed by Djukić et al. [DPL16].

contrast to that, the concepts proposed in this thesis follow the approach of seamlessly integrating with debugging tools for the host language and the visualizations already present for the modelling language. The goal here is to create an unobtrusive, general-purpose debugging tool. The aspect of handling multiple languages and their interactions at once is not covered.

Chapter 3

Used Technologies

This chapter gives an overview of technologies used in the implementation of this project. While some of them are generally suited for implementing model-based debuggers, others are specific to the demonstrator for SCCharts implemented here. Only a brief note on the usage of each technology is given; more implementation details are covered in Chapter 5.

3.1 The Eclipse Platform

The Eclipse IDE is a modular development platform designed to support various languages and development tasks through the use of plugins. This way, a customized application can be assembled to suit the needs of a particular user group without introducing unnecessary performance or disk space overhead through unused features.

The following sections present the extension point system that allows easy integration of new features into Eclipse, the Java Development Toolkit providing IDE support for Java, available debugging features in Eclipse and an overview of KIELER, an Eclipse product for SCCharts development and layout that the demonstrator is based on.

3.1.1 Plugin Infrastructure and Extension Points

Most of Eclipse's functionality is organized in plugins. A large selection of plugins is available online[1], either by themselves or pre-assembled into packages, so-called *features*. Complete ready-to-use applications including all features required for a certain task, called *products*, are also available.

To allow interaction between plugins, Eclipse offers *extension points*. An extension point is an interface provided by one plugin where other plugins can connect if they meet certain requirements. Extension points and the corresponding extensions are defined using the *Extensible Markup Language (XML)*.

For example, a new view can be registered by adding an extension using the extension point `views` provided by the plugin `org.eclipse.ui`. The extension requires a pointer to a class extending *ViewPart* (also provided by `org.eclipse.ui`), a name for the view and a unique ID, as well as other optional fields. The view will then be accessible both within the code (e. g., to include it in a perspective) and to the user (e. g., via the quick access menu, using the name specified in XML). An example of the XML definition can be seen in Listing 3.1.

[1] https://marketplace.eclipse.org

```
<extension
  point="org.eclipse.ui.views">
<view
  category="de.cau.cs.kieler"
  class="de.cau.cs.kieler.sccharts.ui.debug.view.DebugDiagramView"
  id="de.cau.cs.kieler.sccharts.ui.debug.debugDiagram"
  name="Debug Diagram"
  restorable="true">
</view>
</extension>
```

Listing 3.1. An example extension definition, taken from the `de.cau.cs.kieler.sccharts.ui` plugin.

3.1.2 Java Development Toolkit

Eclipse's *Java Development Toolkit (JDT)* provides extensive IDE support for the Java language. Besides editors for Java files, it provides helpful features such as an outline view that facilitates understanding the structure of large code files as well as extension points to programmatically access the *Abstract Syntax Tree (AST)* underlying the code files. Noteably, it also comes with a Java debugging frontend integrated into the IDE, which the demonstrator implementation is largely based on.

3.1.3 Debugging in Eclipse

Eclipse has a language-independent debugging framework that can be expanded using extension points. To adapt this framework for a particular language, several components need to be specified.

The *debug model* is a visual representation of the program structure. For classical imperative languages such as Java, such a model consists of (representations of) threads, stack frames, variables and similar components. An instance of this debug model can then be attached to a program launch in Eclipse, allowing it to visualize the state of that particular launch. Such an attached debug model is referred to as a *debug target*.

The standard Eclipse debug controls can then be used to send commands to the debug target. It is up to the execution environment to react to these commands appropriately. If a custom language interpreter is used, it needs to support suspending, stepping and similar operations by itself.

Breakpoints, Debug Events and Listeners

To facilitate debugging, the Eclipse platform provides a set of central event producers, to which listeners can be attached, following the *observer pattern* [GHJ+94], through the use of extension points.

One of these components is the `BreakpointManager`. It can be used to programatically set and remove breakpoints on resources (such as source code files). When a debug run starts, the BreakpointManager will ensure that all relevant breakpoints are added to the launch and removed when the user removes them. Similarly, when a breakpoint is hit, the `BreakpointManager` will notify all registered breakpoint listeners and suspend the execution if required. The listeners may vote whether the execution should be suspended.

Whenever the state of the debug target changes (through reaching a breakpoint, but also user actions such as clicking the debug controls), *DebugEvents* are fired, which all registered listeners will be informed about. For example, an editor can use this mechanic to know when to update the currently highlighted line.

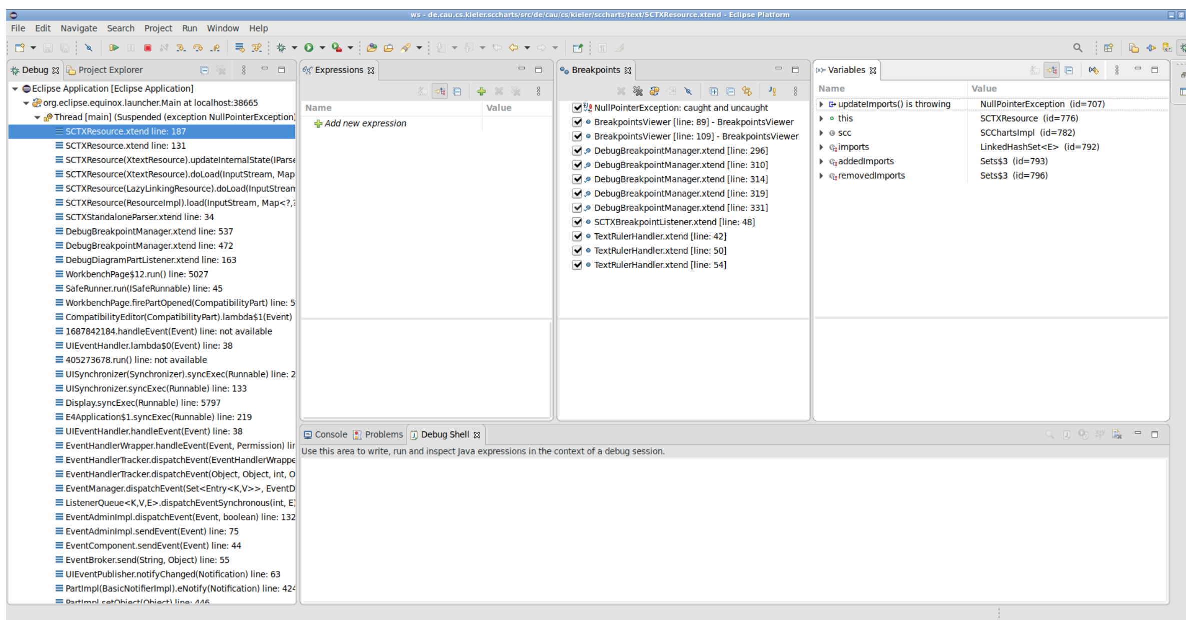


Figure 3.1. Debug perspective in Eclipse. Views rearranged for better presentation.

Other Debugging Tools in Eclipse

Apart from the internal components mentioned above, the UI of Eclipse contains many features geared towards debugging, bundled in the *Debug perspective*. All of these tools can be seen in Figure 3.1.

Debug view (on the left) This view shows the `DebugModel` of the current launch, including the stack state in which the current thread was suspended.

Expressions view (left center) Here, the user can enter expressions in the target language that can be evaluated based on the runtime memory state during debugging. This is helpful to check invariant properties of the program or other conditions.

Breakpoints view (right center) In the breakpoints view, the user can see and control all breakpoints in the current workspace. Each breakpoint can be individually enabled and disabled, deleted or modified. This view also allows the user to specify conditions under which breakpoints should be active to avoid unnecessary suspensions.

Variables view (on the right) This view shows the values of all variables in the current scope. For non-primitive values, clicking the field name will reveal information on the contained fields and their values. This view also allows the user to choose new values for each variable to interact with the runtime state of the program.

Debug Shell (on the bottom) Finally, this shell can display the result of evaluating expressions either entered directly into it or selected in the editor at runtime.

While all of these views are useful in some contexts, the breakpoints and variables views are most relevant to this thesis. Neither the demonstrator nor any of the presented concepts use the remaining components, even though they may become relevant for future work.

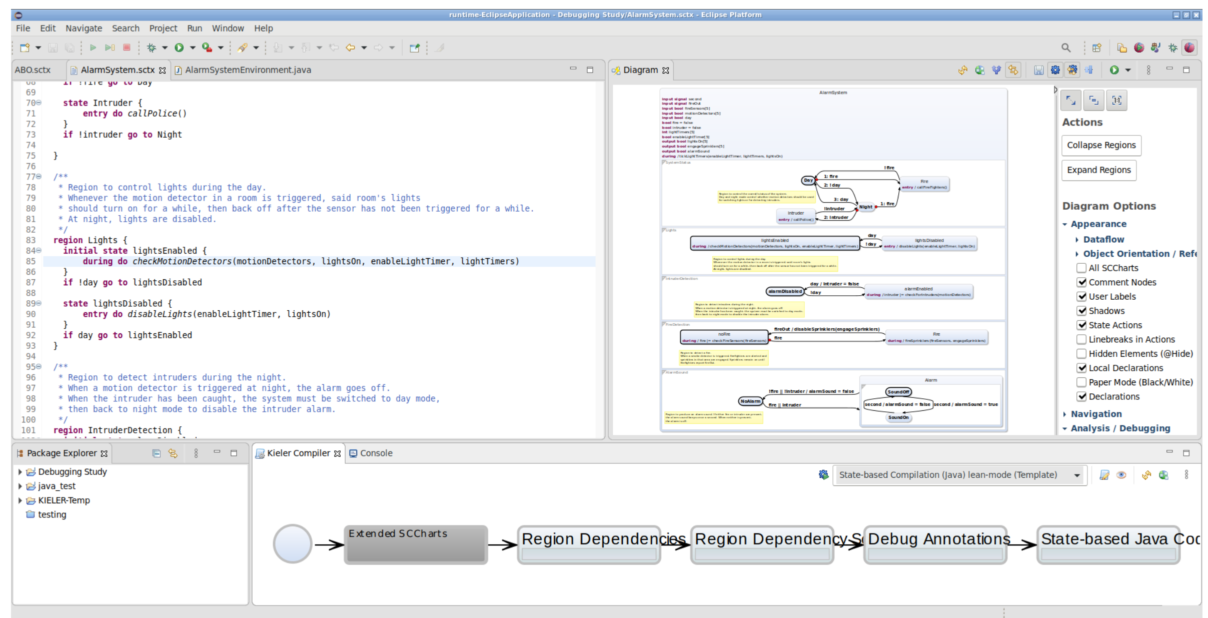


Figure 3.2. A screenshot of the KIELER SCCharts development perspective. It features an editor for SCCharts code, a transient diagram view, and a compiler selection view.

3.2 KIELER

The *Kiel Integrated Environment for Layout Eclipse RichClient (KIELER)*[2] is an Eclipse product developed by the *Real-Time and Embedded Systems* group[3] at Kiel University.

KIELER is centered around the development and simulation of SCCharts as well as automatic graph layout. For these purposes, a fully functional text editor to write and edit a textual representation of SCCharts, diagram synthesis support to transiently display the SCCharts written in the editor and a model-based compiler [SSH18] are provided. The example debugger for the SCCharts language has been integrated with KIELER as well.

Automatic layout of SCCharts and other graphs is provided by the *KIELER Lightweight Diagrams (KLighD)* view framework [SSH13] along with the *Eclipse Layout Kernel (ELK)*[4], which brings a large set of layout algorithms and its own graph language. ELK and KLighD can be used to automatically layout and display a large variety of graphs.

The compiler offers modular *model-to-model (m2m)* transformation systems to compile SCCharts models to a set of host languages including C, Java and VHDL. Due to the modular nature of the compilation, it is straightforward to add processors or entire compilation chains for custom applications. A screenshot of the KIELER SCCharts development perspective can be seen in Figure 3.2. The following sections discuss the basic functionality of the compiler required for this thesis; a more detailed breakdown can be found in [SSH18].

[2]http://rtsys.informatik.uni-kiel.de/kieler
[3]http://www.rtsys.informatik.uni-kiel.de
[4]https://www.eclipse.org/elk

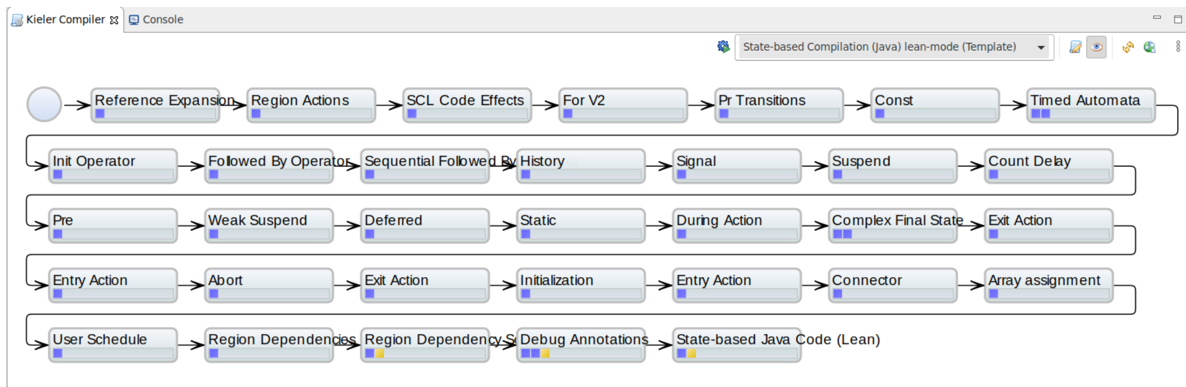


Figure 3.3. The KIELER compiler selection with the state-based compilation system and intermediate results.

3.2.1 Modular Compilation Systems

A key feature of the KIELER compiler is the modular compilation concept. Developers can implement *processors*, which are able to perform a single m2m transformation step (e. g., removing all instances of a specific high-level SCCharts feature and replacing them with appropriate lower-level constructs). Input and output model types may differ (e. g., for the final code generation step), but most processors will typically perform transformations within the same model type.

Using a simple DSL, multiple processors (and other systems) can be assembled into a *compilation system*. This way, subsystems can be defined and reused in multiple compilation systems, making it easy to combine various frontends with multiple backends.

Based on the input and output types of the processors, KIELER offers only the matching systems when compiling a certain type of model. The compilation process itself executes all processors sequentially and stores each intermediate model. This way, the user can inspect the different states and transformations the process went through. The KIELER compiler selection with the state-based compilation chain selected can be seen in Figure 3.3. Intermediate models can be accessed by clicking the blue boxes in each compilation step.

3.2.2 Model Tracing

Apart from saving and displaying the intermediate transformation steps, the compiler also allows for *tracing*. Processors that support tracing will offer a mapping between original and generated model elements, i. e., each generated model element can be mapped back to the original element it was generated from. If the entire compilation chain supports tracing, it is therefore possible to determine the source model element for every component in the transformed model and vice versa.

It is important to note here that the tracing information is provided by the compiler environment and is thus only available during compilation and only to the processors that are part of the compilation chain currently being executed. If tracing information is required after the compilation finishes, e. g., at runtime, it needs to be persisted elsewhere.

Chapter 4

Design and Concept

This chapter presents debugging concepts for model-based languages. As stated in Chapter 1, the main goal is to allow debugging of generated code without the need for the user to manually navigate it. While many concepts presented here can be applied to various model-based languages, some are exclusive to statechart-like languages and others must be adapted quite specifically to the respective use case. SCCharts is used as an example modelling language where appropriate.

As mentioned in Section 2.2, the demonstrator implementation does not cover all possibilities discussed here and, while aiming to be as modular as possible, it only covers a single SCCharts compilation system, namely the state-based compilation, as presented in Section 2.2.1, for Java. Chapter 5 gives implementation details and highlights places where other actions need to be taken for other host languages. Concepts for debugging SCCharts code generated using other compilation approaches can be found at the end of this chapter.

Section 4.1 describes the process of defining language-specific semantics for breakpoints, which is the base of the automatic breakpoint placement discussed in Section 4.6. In Section 4.2, approaches for persisting tracing information for use at runtime are discussed while Section 4.3 describes how the source model can be made available to the debugger at runtime. Design choices for the debugger's user interface, particularly for graphical languages, are discussed in Section 4.4.

After these rather general sections, the concrete implementation requirements are presented using the SCCharts demonstrator as an example. Section 4.5 discusses how the requirements to the code generator can be realized for an SCCharts compilation chain. In Section 4.6, the process of finding the correct breakpoint locations according to the desired semantics is shown before Section 4.7 describes how relevant runtime information can be extracted from the debugging environment to be shown to the user. Finally, Section 4.8 and Section 4.9 give ideas on how the demonstrator could be adapted for other SCCharts compilation approaches as futher examples.

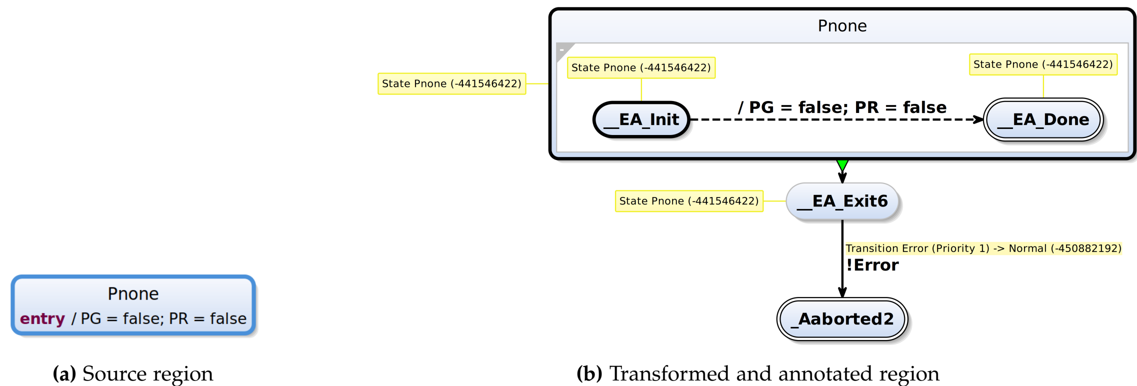
4.1 Breakpoint Semantics

As described in Section 1.4.1, the tool needs to automatically determine appropriate code locations to set breakpoints associated with model elements. However, finding these locations is not trivial, especially since they depend on the desired semantics for model-level breakpoints.

This decision must be made for each language individually since the types of breakpoints depend largely on the language's model of computation as well as the use cases of the language. This section discusses possible semantics for breakpoints associated with different SCCharts model elements as an example.

4.1.1 State Breakpoints

For breakpoints in text-based languages, it is common to see them triggered each time a line of code is executed. If a method with a method breakpoint is called multiple times or a loop with a line



(a) Source region

(b) Transformed and annotated region

Figure 4.1. Original and transformed model extract. A total of four states are generated from state `Pnone` in the original model.

breakpoint in it runs repeatedly, the breakpoint is triggered each time. If that behavior is not desired, many debuggers offer options such as conditional breakpoints (which are only triggered when a certain expression evaluates to `true`) or hit counts (which will only enable the breakpoint after being hit x times).

Following a similar approach, implementing breakpoints on SCCharts states as simple line breakpoints inside the method(s) generated for a state (or a method breakpoint on the method(s)) seems to be reasonable. However, this leads to multiple disadvantages.

Firstly, since multiple methods may be generated from a single source-level state due to new states being introduced during the m2m transformations performed by the compiler, this approach leads to multiple breakpoints being triggered per state. For the example shown in Figure 4.1, a breakpoint placed on the source state `Pnone` leads to four breakpoints being placed, one on each of the methods corresponding to the states generated from `Pnone`. This causes multiple suspensions and the user having to hit `resume` four times even though nothing changed for them.

Secondly, since SCCharts are not necessarily intended to be used in an event-driven way (i. e., a tick is performed every time an input event arrives), but may also be used in a time-driven way (i. e., a tick is performed every x milliseconds)[1], suspending the execution every time the method associated with a state is executed potentially leads to many suspensions where the program state has not changed at all compared to the last suspension.

Therefore, the semantics of state breakpoints in the demonstrator is to suspend the execution *once* as soon as the state is entered. All following ticks where the state associated with the breakpoint remains active do *not* cause a suspension. However, as soon as the state is left, the next time it is re-entered (even if that occurs within the same tick), the breakpoint is triggered again.

This way, the user can track when the state is entered and therefore get an understanding of how the SCChart transitions between states without unnecessary suspensions when nothing changes.

4.1.2 Transition Breakpoints

Apart from breakpoints on states, one may want to place breakpoints on transitions as well. Even though observing when a transition has been taken by placing a breakpoint on the transition's target

[1]Timed SCCharts [SHM+18] are an exception, where the time between ticks can be adjusted at runtime depending on the frequency of expected input events. Here, one could argue that suspending in every tick would in fact be desirable.

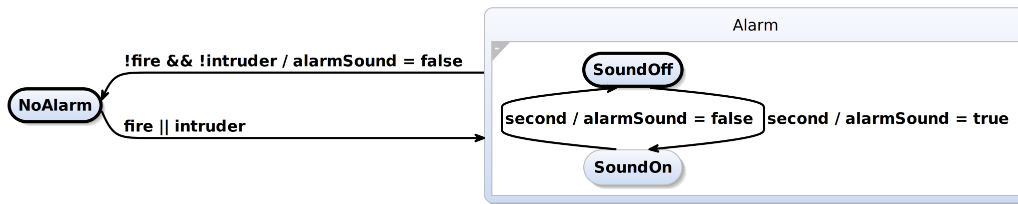


Figure 4.2. `AlarmSound` region from the `AlarmSystem` SCChart.

state is possible in theory, this becomes confusing as soon as the target state has multiple incoming transitions. At the same time, this method only allows to see that a transition has been taken after the fact, without allowing the user to observe the evaluation of the transition's guard or even changing some of its components at runtime.

From this idea, two different possibilities for breakpoint semantics arise. Both appear equally valid, even though their applications lie on two different levels of granularity. A user may either be interested in the transition being *taken*, i. e., the effect being executed and the target state becoming active, or the transition being *checked*, but not necessarily taken, i. e., the guard being evaluated.

The first case may seem more practical since it follows the idea of breakpoints tracking how the SCChart transitions between states described above. However, it may well happen that the SCChart is in a certain state and the user expects it to take a certain transition, but that does not happen. In that case, neither of the types of breakpoints introduced so far help since neither of them is triggered when the source state has already been entered, but the outgoing transition is not taken. Considering the example in Figure 4.2, a user may either be interested in the moment the alarm is triggered (i. e., the transition from `NoAlarm` to `Alarm` being taken) or perhaps in examining each component of the guard when the alarm does not turn off when expected to (i. e., the guard on the transition from `Alarm` to `NoAlarm` evaluating to `false`).

To allow the user to observe the evaluation of the transition's guard or even influence it, a second type of transition breakpoint is introduced. Note that it should be possible to place both types of breakpoint on the same transition without them interfering.

Transition Taken Breakpoints

As described above, this type of breakpoint is triggered *once* when a transition is taken, i. e., its guard evaluates to `true`. Since multiple actions may happen during the transition (e. g., multiple effects, the target state being set, ...), the breakpoint suspends on the *first* one of these actions to be executed. When the transition is taken a second time (even within the same tick), the breakpoint is triggered again.

Transition Check Breakpoints

This type of breakpoint is triggered *every time* the transition's guard is evaluated, regardless of when the state has been entered or whether the transition is taken. This way, the user can observe each component of the guard expression precisely at the time the guard is evaluated to detect issues. As described previously, this type of breakpoint may lead to many suspensions with no changes in between them.

```
/**
 * State Pnone (-441546422)
 */
private void TRAFFIC_LIGHT__EA_Exit6(TRAFFIC_LIGHT_regionPedestrianContext context) {
  if (context.delayedEnabled && (!iface.Error)) {  // Transition Error (Priority 1) -> Normal (-450882192)
    context.delayedEnabled = false;
    context.activeState = TRAFFIC_LIGHT_regionPedestrianStates._AABORTED2;
  } else {
    context.threadStatus = ThreadStatus.READY;
  }
}
```

Listing 4.1. Example code with generated marker comments. Names shortened for better readability.

4.1.3 Other Breakpoints

The three types of breakpoints described above have been implemented in the demonstrator. More types of breakpoints would bring an increase in flexibility when debugging, but also require more training for the users.

The required types of breakpoints depend largely on the features most used in a model. For example, some models may rely largely on core SCCharts features such as states and transitions, using few advanced constructs. For these models, it is preferrable to have few types of breakpoints to reduce complexity of the UI and make the tool simpler overall. However, a user relying heavily on actions may find that not being able to place breakpoints on them hinders debugging. Therefore, adding a new type of breakpoint to place on entry, exit and during actions may be desirable in the future. However, the evaluation studies described in Chapter 6 suggest that the three types of breakpoints presented above may be sufficient.

4.2 Markers in Generated Code

Section 3.2.2 introduced the concept of compiler tracing. With this feature, it is possible to easily determine what model elements in the transformed model were generated from what source elements. This information is required later since the generated code usually allows only a mapping back to the transformed model, not to the source model.

However, as mentioned earlier, the tracing information is only available during the compilation process, not at runtime when it is needed. Therefore, it has to be persisted. This is done using comments in the generated code which are either attached to methods in the form of Javadoc comments or added to certain lines as end-of-line comments. A detailed description of how these comments can be created during the code generation process can be found in Section 4.5. A code excerpt with such marker comments for a state and its transitions can be found in Listing 4.1.

The idea behind these marker comments is to create an association between source model element and code location, allowing both the placement of breakpoints in the generated code and the extraction of runtime information in relation to the source model. Without these markers, there would be no way to determine that the method in the example above was generated from the entry action of state `Pnone` of the original model.

While some may consider it better style to use Java annotations for this purpose, using comments brings some advantages. Firstly, they are not only useful for the tool to determine where to place breakpoints and to interpret them later on, but also for the user when reading the generated code.

Thanks to the comments, the user can now easily see what methods are generated from what source elements and thus better understand the structure of the code. Secondly, and perhaps more importantly, comments do not influence the runtime behavior of the code outside a debugging environment. As stated in Sections 1.4.4 and 2.4, interference with the code's runtime behavior needs to be kept to a minimum, which annotations cannot guarantee since they will always be active at runtime. Another advantage of using comments over annotations is that the concept can be transferred more easily to other target languages such as C, where annotations may not be available, but comments are.

4.3 Source Model Access

The previous section discussed how to preserve the tracing information until it is needed at runtime to map the generated code to the original model. However, for this to be possible, access to the source model itself is also required. This can be achieved by placing a string variable containing a file path pointing to the original source file in the generated code at compile time.

Since developer teams usually use a clearly organized project structure and someone debugging code generated from a source model usually has access to the original source file, it seems reasonable to assume that the source model is available at runtime from a location already known at compile time. The model file can then be loaded at runtime using the standard model parser.

In case the source file was unavailable at runtime, the compilation chain could easily be altered to include not only the file path, but the entire file content in the generated code and parse the model from there using a standalone parser. This option would eliminate the need for the original model file, but at the cost of introducing a larger string variable into the code, resulting in potentially decreased performance at runtime for large models.

Following the reasoning presented in the previous section and Section 1.4.4, it would be ideal to be able to remove the path variable as well and replace it with a comment to minimize the impact on runtime behavior. For breakpoint placement and extraction of runtime data, using comments is viable since the editor containing the source code is always open when either occurs. However, there are instances where the original model must be retrieved without the editor being open and thus without access to the source code, therefore comments cannot be used here. When the path is located in a String variable in the class under debugging, it can be extracted from the runtime stack. Details on when this occurs can be found in Section 5.6.

4.4 User Interface

To make a debugger as intuitive as possible even for new users, a well-designed *User Interface (UI)* is key. This refers both to the useability of the tool without much instruction and the newly introduced components fitting in with the design of the development environment and the current workflow.

When integrating a new debugger with an existing IDE, it is therefore advisable to use existing debuggers for that IDE as a guideline. For this reason, many of the Eclipse debug controls already present (i. e., buttons for starting / stopping / resuming a debug run, stepping etc.) have been reused without changes in the demonstrator to make the user feel at home. The remaining components have been designed to fit in with KIELER and the SCCharts development process. The user interface with the relevant components marked can be seen in Figure 4.3.

Chapter 6 presents two small studies conducted with both professional SCCharts developers and computer science students that tested the demonstrator and were asked to evaluate, among other

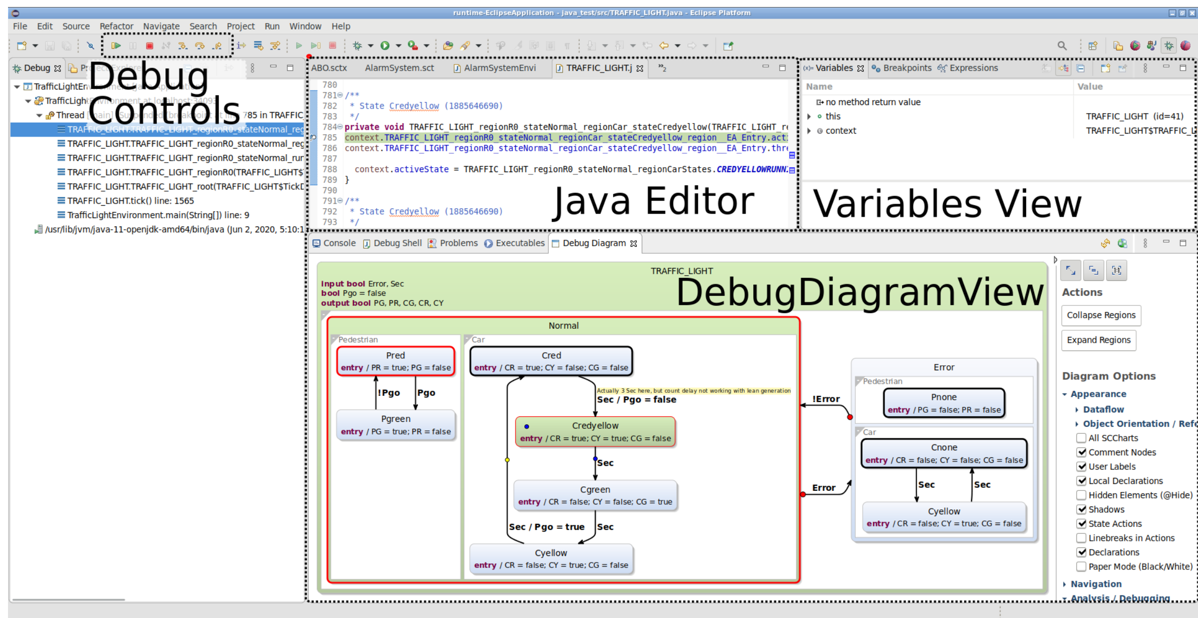


Figure 4.3. The demonstrator's user interface with components relevant to debugging highlighted.

criteria, the ease of use and intuitiveness of the UI design. According to these studies' participants, the design goals have been largely met.

4.4.1 Debug Diagram View

Depending on the language and the desired debugging workflow, it may not be necessary to introduce any new views at all. As an example, debugging Java code in Eclipse does not require any Java-specific components since all necessary functionality is already provided by the Eclipse platform and its debugging plugin. Especially for text-based imperative languages, using these components makes sense.

However, for graphical languages such as SCCharts, custom views are required to display visual debugging information in addition to that provided by the standard Eclipse views. When debugging generated code, the views provided by the platform display host-language-level information on the execution; to add a layer of abstraction and display the information on the model level, other means are required.

Therefore, the demonstrator introduces a new UI component, the *DebugDiagramView*. This view is based largely on KIELER's KLighD view [SSH13], but bears some key differences, which are summarized in Figure 4.4. The regular KLighD view is transiently linked to active SCCharts editors, meaning that it always displays the model being edited there, or a placeholder message if the active editor does not contain an SCChart. The view diverts from this behavior if intermediate models from the compiler view are selected or during simulation, in which case it displays the selected model regardless of editor, possibly with some highlighting in the case of the simulation.

While it would be possible to add the mode of displaying the model associated with a generated code file in a host language editor to the same view, this may lead to confusion since there is no way of distinguishing between the model being displayed for the currently open SCCharts editor and the same model being shown for the generated code during debugging.

Scenario	KIELER Diagram View	DebugDiagramView
Active SCCharts Editor	displays SCChart	empty
Active Java editor (generated code)	empty	displays SCChart
Active Java editor (other code)	empty	empty
Intermediate compiler result selected	displays intermediate model	empty
Running simulation	displays SCChart with runtime information	empty
Running debug session	empty	displays SCChart with runtime information

Figure 4.4. Comparison of use cases for diagram views in KIELER.

Also, allowing the user to set and remove breakpoints by clicking model elements in the view requires changes to the diagram synthesis, which are detailed in Section 5.5.2. It is not trivial to detect which of the two types of model is being displayed, and thus whether the current model needs the adapted synthesis.

Since the regular KLighD view's functionality of interacting with SCCharts editors, the compiler and the SCCharts-level simulation is not required when debugging generated code, the KLighD view can be used as before during development, while the new Debug Diagram View is used in the debug perspective. This way, the user will never need to have both views open at the same time.

4.4.2 Setting Breakpoints

To circumvent the need for the user to set breakpoints in the generated code, an option to set breakpoints on the model level is required. These breakpoints must then be coupled to host language-level breakpoints according to the breakpoint semantics as described in Section 4.1.

For textual languages, using a double click next to the model line where a breakpoint should be placed follows a common way of setting breakpoints and will therefore be intuitive for developers with experience in debugging other textual languages. For graphical languages such as SCCharts, a similar approach can be used by allowing the user to set and remove breakpoints by clicking model elements in the graphical debug view mentioned above.

This approach has been chosen for the SCCharts demonstrator, even though the language is used with a text editor and automatically synthesized visual representations. The main advantage is that using the newly introduced `DebugDiagramView` eliminates the need for an SCCharts editor displaying the source model during debugging. To allow the user to set breakpoints on the source code, which they may be more familiar with, an additional mapping between generated code, displayed diagram and the source model code would be required to allow the user to quickly identify which source-level breakpoint has been hit to allow them to interact with it seamlessly. Since there is no other use for the source model code in the debugging scenario and the Debug Diagram View is already available anyway, directly clicking model elements to set and remove breakpoints is the adequate way.

To set or remove a breakpoint on a state, the user can double-click anywhere on the state, much like one would double-click next to a line of source code to place a breakpoint. For transitions, Section 4.1.2 lists two different types of breakpoints, which require separate keybindings. Since the `TransitionTakenBreakpoint` is semantically more similar to the `StateBreakpoint` and is considered the default transition breakpoint, it can be placed and removed through double-clicking a transition. For the `TransitionCheckBreakpoint`, the behavior is more intrusive and there are fewer use cases for it,

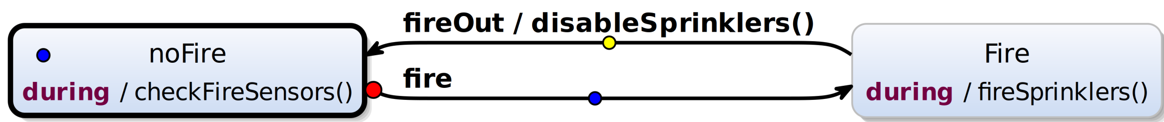


Figure 4.5. Different types of breakpoints as displayed by the Debug Diagram View.
Note that the red circle indicates a Strong Abort transition and is not related to debugging.

therefore it can only be placed by holding shift while double-clicking a transition.

4.4.3 Visual Semantics

An essential part of debugging is to examine the runtime state of a program while it is suspended. While Eclipse and other IDEs already offer a selection of debugging components, which were presented in Section 3.1.3, most of them are hardly suited for displaying information graphically. The newly introduced `DebugDiagramView` and similar components can fill this role, however, they need a visual semantics that clearly communicates the desired information even to users new to the language.

Breakpoint Markers

As explained in the previous section, the view can be used to set and remove breakpoints in the source model. To indicate whether a breakpoint has been set on a model element, clear markers are required. The demonstrator uses circular markers similar to those used in common debuggers for text-based languages to indicate line breakpoints.

For states, the markers are placed in the top left corner, with some distance to ensure they do not collide with the rounded corner and be well visible. They use a dark blue color to follow the blue-themed color scheme of SCCharts and to resemble common debugger breakpoints. In Figure 4.5, a `StateBreakpoint` can be seen on the state `noFire`.

For transitions, breakpoints are placed on the graph edge near the middle of the transition. They cannot be placed precisely in the middle since both types of breakpoints may be placed on the same transition, requiring them to fit next to each other. To indicate similar semantics, `TransitionTakenBreakpoints` use the same marker as `StateBreakpoints`, while the `TransitionCheck-Breakpoint` uses a bright yellow marker to clearly distinguish the two types. Examples of each marker can also be found in Figure 4.5.

If the visual syntax of the language already contains similar constructs, markers should be chosen that can clearly be distinguished from those constructs. SCCharts contain strong abort transitions, indicated by a red circle at the beginning of the transition, however the clear difference in color, size and placement between those circles and breakpoint markers allow for differentiation. Finding out whether using differently shaped markers to further avoid confusion outweighs the disadvantage of less resemblance with traditional line breakpoints would require a study outside of this thesis.

Highlighting

While setting and removing breakpoints is possible at any time the matching generated code editor is open, regardless of whether a debug session is currently running, the `DebugDiagramView` displays additional information during debug sessions. Figure 4.6 shows an example diagram during debugging.

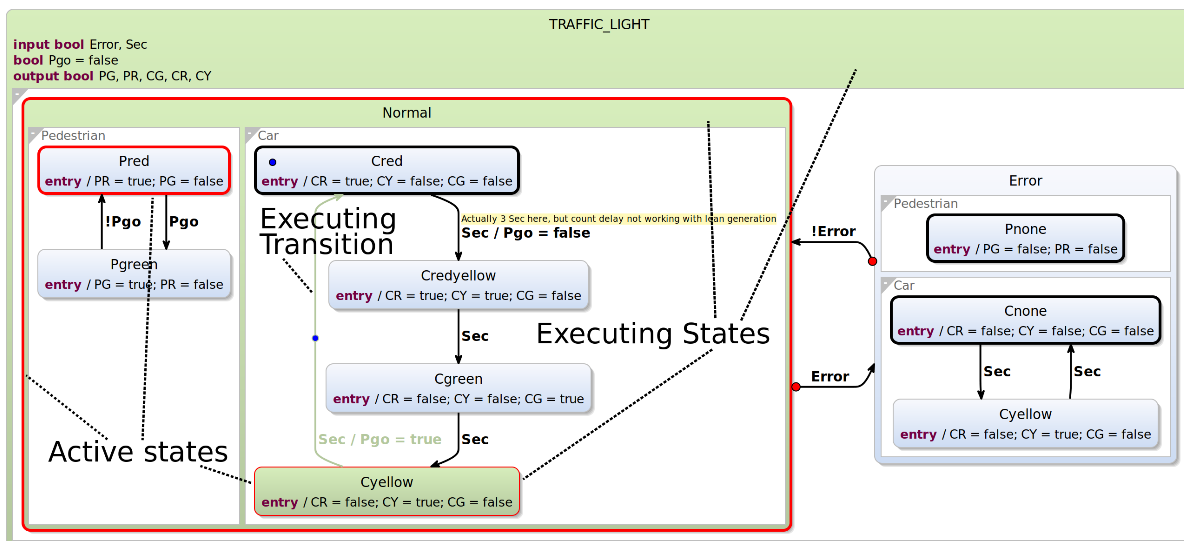


Figure 4.6. An SCChart during a debug run. Red frames indicate active states, green backgrounds mean executing elements.

Depending on the language's model of computation, a variety of information can be available and relevant at a given time. For example, debuggers for visual dataflow languages such as *SCADE*[2] may display the equation currently being evaluated along with current variable values. On the other hand, debugging state-machine-based languages such as SCCharts profits from highlighting active states and transitions, especially in concurrent contexts.

Since SCCharts may contain concurrent regions, which are sequentialized during state-based compilation as described in Section 2.2.1, the generated code always runs in a single thread and thus, only one method can be currently running when the execution is suspended. However, the regions' `context` datastructures always keep information on active states even for currently inactive regions. A more detailed description of how runtime information is extracted in the demonstrator can be found in Section 4.7.

If there are multiple parallel regions in the current scope, there may therefore be multiple *active* states (i. e., states the SCChart is currently in), but only one of them can be *executing* (i. e., one of its associated methods is running). For surrounding scopes, the same rules apply iteratively, where superstates are considered to be executing when one of their associated methods is currently on the callstack, even if it has called another (substate's) method.

Since both of these classes of states are relevant to understanding the current state of the model as a whole, they need to be highlighted in the diagram in a way that makes them clearly stand out from non-highlighted states while minimizing the potential for confusion between the two.

To indicate that a state is *active*, it receives a *red outline*. Red is chosen here since the highlighting of active states in the KIELER simulation uses the same color, so experienced SCCharts developers will recognize the semantics conveyed by the outline. Since the root state will always be active when the generated code is executing, the red outline is omitted to reduce visual clutter.

For *executing* states and other model elements, a *green background* was chosen, with a color closely resembling the one used by Eclipse when highlighting the currently executing line of code in an editor. Again, experienced developers can immediately associate the color with the current instruction

[2]https://www.ansys.com/en-gb/products/embedded-software/ansys-scade-suite

pointer. Here, the root state also receives a green background even though it is always executing when a breakpoint within the SCChart is hit. This is to make sure that there is a clear visual difference between the diagram while debugging and while no session is active, even if the currently active and executing states within the diagram are very small or otherwise hard to spot.

It is important to note that even though red and green may be hard to distinguish for some users, the fact that one is used for background and the other one for foreground coloring should still make it possible for those users to use the tool. This tradeoff has been accepted in favor of the advantages listed above.

Figure 4.6 shows all of these highlightings in practice. State `Normal` is active and executing, as well as its substate `Cyellow`. Since the `TransitionTakenBreakpoint` between `Cyellow` and `Cred` has been hit, the transition is currently executing and thus highlighted, too. In the parallel `Pedestrian` region, no states are currently executing due to the sequentialization, but state `Pred` is active nonetheless.

4.5 Changes to the Compilation Chain

The previous sections outlined requirements and concepts for debugging model-based languages in general. This one and the following sections are closer to the demonstrator implementation and put more focus on SCCharts and KIELER since the approaches are closely tied to the language and the debugging environment. Technical details of the implementation can be found in Chapter 5.

This section focuses on integrating the required marker comments and source model references into a state-based compilation system, using the state-based Java code generation for SCCharts as an example.

4.5.1 Adding Debug Markers

Since the tracing information is only accessible during compilation, the compilation chain needs to be extended by an additional step or processor to support debugging. This processor runs as the last one on model level before the actual code generation. This way, it can be ensured that all elements used in the code generation are present already when this step runs and no model elements can be introduced afterwards.

Using Tracing to Determine Source Elements

The processor relies on the tracing mechanic described in Section 3.2.2 to determine the source model elements for all generated ones. This mechanic may be supported in different ways, depending on the compiler. The KIELER compiler's environment provides a mapping between source and transformed model elements or vice versa, provided that all processors in the compilation chain support tracing, i. e., correctly register the alterations they make in the compiler environment.

Since no m2m transformations in KIELER ever create a single model element from multiple source elements, the mapping from transformed to source elements will always yield a single source for each transformed model element (or possibly none if the tracing chain is incomplete). For other languages or compilation systems where this is not the case, the following concepts need to be adapted accordingly, e. g., by including a list of model elements instead of a single one.

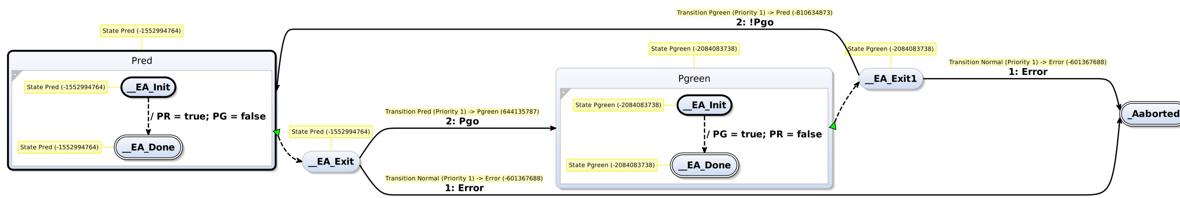


Figure 4.7. A transformed and annotated SCCharts model. Comments attached to model elements are displayed as yellow boxes.

Annotations and Comment Format

Once the source model element has been determined for an element in the transformed model, the processor can attach a comment to the transformed element, much as a user could in the source code. In the case of SCCharts, the annotated model can be examined as an intermediate compiler result; an example extract from a transformed and annotated model can be found in Figure 4.7.

If there are user comments already present, the new comment annotation is added last. In contrast to the tracing data, these comment annotations are not part of the compiler environment, but of the transformed model, and will thus remain available.

The comments use a specific format expected in later steps that was designed to be machine-readable, but also helpful for a human reader looking at the generated code. Most importantly, the comments must uniquely identify each model element. Since in many languages, different model elements may have the same name as long as they reside in different scopes, simply annotating the source model element's name is not sufficient. However, including the name makes it easier to determine a selection of potential matches before comparing other, more unique identifiers. An added benefit of including the actual name is that it makes reading the code much easier for humans.

Apart from the source model element's name, the comments contain a hash over a fully qualified name of a model element, including all parent scopes' names. Since no two model elements may have the exact same hierarchy of parents as well as the same name, this ensures unique hashes for each element[3].

The exact comment format used to mark SCCharts states and transitions in the demonstrator implementation is described in Section 5.3.

4.5.2 Template Adaptations

After the m2m transformations are complete, code is generated from the transformed model in the last compilation step. For SCCharts, this step is template-based and changing the template is sufficient; for other approaches, the compiler must be adapted accordingly. To make the tracing information available at runtime, the code generator needs to include the marker comments added in the previous step.

For model elements that have methods associated with them, such as the states in SCCharts, a leading comment containing the comment annotations of the respective model element can be added to the method. This ensures that the marker comments can always be found in a pre-defined place.

For other model elements, such as transitions in SCCharts, there may be multiple lines of code generated from them within a method. In SCCharts, these lines are enclosed within an `if`-statement, with the condition being the transition's guard and the body containing its effects. Since comments

[3]Technically, random *hash collisions* may occur. However, using a modern hashing algorithm and considering the comparably low number of model elements in most practical models, the probability of a hash collision is negligible.

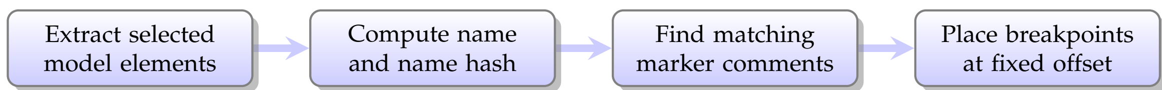


Figure 4.8. Steps to placing a breakpoint corresponding to a model element selected by the user.

cannot be attached here, they are added after the `if`-condition marking the transition's guard as end-of-line comments instead. For other model elements with no such block structure, an appropriate location must be determined according to the language-specific breakpoint semantics.

Additionally, if enums are used to store the active state, they must be extended by a String field containing the same value that is used to mark the state's methods. This way, it is possible to extract the active source model states from the model's internal storage.

4.6 Finding Breakpoint Locations

When the user double-clicks a model element in the diagram view, breakpoints must be set in the appropriate code location(s) automatically. After Section 4.1 introduced the desired semantics for the breakpoints and Section 4.5 described how marker comments are placed in the generated code, this section presents an implementation concept for finding the correct breakpoint locations for a model element. A high-level overview of the concept can be seen in Figure 4.8.

It is important to note that since breakpoints are set by clicking the diagram view, the source model being displayed there is thus already available. And since the diagram view always displays the model associated with the active editor, this in turn means that whenever the user sets a breakpoint on a model in the diagram view, the active editor contains the corresponding host language code, which makes it easy to locate the appropriate file to set breakpoints in. Even though the breakpoint types used as examples in this section are SCCharts specific, the general concept can be applied to all model-based languages where the code generation follows a similar state-based pattern.

4.6.1 State Breakpoints

Since the source model is available and the state that is clicked can be identified by the KLighD framework, the state's full name hash can be computed. Using regular expressions, all comments matching the marker comment pattern described in Section 4.5 can be located. Knowing the specific template format and that the marker comment is always added last, the first line of the method body is always located a fixed number of lines after its marker comment, which allows the tool to place a breakpoint there for each method associated with the given state. In the example shown in Listing 4.2, a user clicking state `Pnone` in the diagram view can expect the tool to place a breakpoint on line 5, since that line is located exactly three lines after the marker comment unambiguously marking the shown method as pertaining to `Pnone`. The specific distance of three lines between marker comment and desired breakpoint location depends on the code generation and desired breakpoint semantics and must be adapted for each compilation chain individually.

To implement the state breakpoint semantics described in Section 4.1.1, the breakpoints must be registered by state so that whenever a `StateBreakpoint` is triggered, said breakpoint as well as all other `StateBreakpoints` associated with the same state can be disabled.

However, re-enabling the breakpoints when the state is left is not as trivial since taken transitions do not usually trigger any events visible to the debugger. Therefore, artificial `TransitionWatch-Breakpoints` are added to all of the state's outgoing transitions. The placement of these breakpoints

```java
/**
 * State Pnone (-441546422)
 */
private void TRAFFIC_LIGHT__EA_Exit6(TRAFFIC_LIGHT_regionPedestrianContext context) {
  if (context.delayedEnabled && (!iface.Error)) {  // Transition Error (Priority 1) -> Normal (-450882192)
    context.delayedEnabled = false;
    context.activeState = TRAFFIC_LIGHT_regionPedestrianStates._AABORTED2;
  } else {
    context.threadStatus = ThreadStatus.READY;
  }
}
```

Listing 4.2. Example code with generated marker comments. Names shortened for better readability.

is the same as for `TransitionTakenBreakpoints`, which is described in detail in the following section. However, a `TransitionWatchBreakpoint` does not need to suspend the debug session when hit, nor be displayed to the user in the diagram view. Breakpoint listeners are still notified, however, and can thus detect that the transition has been taken and its source state must have been left. With this knowledge, all `StateBreakpoints` associated with the transition's source state can be re-enabled.

4.6.2 Transition Breakpoints

Setting transition breakpoints of either type follows a similar pattern as for state breakpoints. From the known model, selected transition and editor, a regular expression can be used to find all code locations associated with the given transition. Even though there usually is just one marker for simple transitions, some cases (e. g., strong abort transitions) may cause multiple code locations to be generated from a single source transition. Listing 4.2 shows such a case: The transition marker in line 5 belongs to a strong abort transition in the original model, going from state `Error` to state `Normal`. Lines of code generated from this abort transition appear in every child state of `Error`, which `Pnone` is one of.

From the structure of the template used in code generation, it is known that the marker comment is always located on the same line as the transition's guard expression. Therefore, `TransitionCheck-Breakpoints` can be placed on each line with a matching marker comment, so line 5 in our example. Accordingly, `TransitionTakenBreakpoints` (and `TransitionWatchBreakpoints` mentioned in the previous section) are placed one line below the marker, so on the first line of the `if`-statement's body. This way, the `TransitionCheckBreakpoint` is triggered whenever the `if`-statement's guard is evaluated, but the `TransitionTakenBreakpoint` only triggers when the transition's body is run. Here, it is worth noting that even if the transition does not have an effect, there will still be at least one line in the body to set the context's active state to the transition's target state, making this method safe for those cases, too.

4.7 Retrieving Runtime Information

Whenever a debug session is suspended on a breakpoint introduced by the debugger, a collection of data on the currently running session is displayed to the user as described in Section 4.4.3. How the highlighting of active and executing model elements is implemented is presented in Section 5.5; this section describes how the different pieces of information can be extracted from the runtime environment. While presented specifically for SCCharts here, other model-based and host languages can be analyzed in the same way, as long as the runtime information is accessible in a similar way to what is described here.

```
/**
 * The runtime thread data of region
 */
public static class TRAFFIC_LIGHT_regionR0Context {
  ThreadStatus threadStatus;
  TRAFFIC_LIGHT_regionR0States activeState;
  TRAFFIC_LIGHT_regionCarContext TRAFFIC_LIGHT_regionCar
      = new TRAFFIC_LIGHT_regionCarContext();
  TRAFFIC_LIGHT_regionPedestrianContext
     TRAFFIC_LIGHT_regionPedestrian = new
     TRAFFIC_LIGHT_regionPedestrianContext();
  [...]
}
```

Listing (4.3) Context object generated for a region. Names shortened for better presentation.

```
/**
 * Enumeration for all states of the  region
 */
public enum TRAFFIC_LIGHT_regionR0States {
  NORMAL("State Normal (1131102381)"),
  NORMALRUNNING("State Normal (1131102381)"),
  ERROR("State Error (998096258)"),
  ERRORRUNNING("State Error (998096258)"),
  __EA_INIT18("State TRAFFIC_LIGHT (-934068026)");

  private String origin;
  TRAFFIC_LIGHT_regionR0States(String origin) {
      this.origin = origin;
  }

  public String getOrigin() {
      return origin;
  }
}
```

Listing (4.4) Generated enum of a region's states

Figure 4.9. Code extracts from `TrafficLight` SCChart.

4.7.1 Active States

As presented in Section 2.2.1, there is a `Context` structure in the generated code for each region in an SCChart containing the currently active state. One such structure can be seen in Listing 4.3. The root context is always stored in a variable with a pre-defined name, so this variable can be extracted from the runtime heap of the thread that has been suspended. This context structure (just as any non-root ones) contains one `threadStatus` field (line 5), one `activeState` field (line 6) and any number of contexts representing the state of subregions.

The `threadStatus` field contains one of four possible values, which can be used to determine whether the region has been initialized yet or whether it has already terminated. Both for unitialized and terminated regions, the active state of it and all of its subregions should be disregarded. These regions occur within states which are not currently active and do therefore not have an active state.

In the `activeState` field, an enum value pertaining to the currently active state of the region is stored. An example enum for the root region of the `TrafficLight` SCChart can be seen in Listing 4.4. Since the adapted template added a marker string to these enum values, they can then be used to find the corresponding model element to highlight.

Any other field can be assumed to contain the context for a subregion, which is recursively searched for active states to be highlighted.

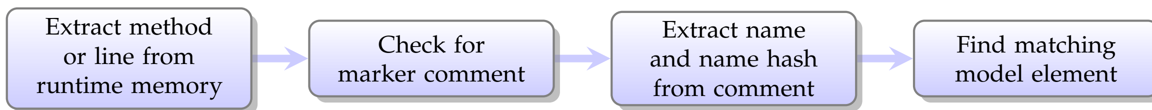


Figure 4.10. Steps to extracting executing model elements from the program's runtime memory.

4.7.2 Executing States

While the active states can be read directly from memory, determining the executing states requires the use of the callstack's structure. Starting at the topmost stack frame, each method on the stack is analyzed and if it is generated from a state, said state is considered to be executing.

To determine whether the method was generated from a state, the fact that most Eclipse editors provide an *Abstract Syntax Tree (AST)* representation of their content can be exploited. From the AST of the active editor, all methods are retrieved that have the same name as the one the current stack frame belongs to. After finding the method, the attached comment can be parsed and mapped to a source model state with the same method as above. Whenever a method appears on the callstack that cannot be found in the current editor or does not have a comment matching the expected marker format, it is ignored.

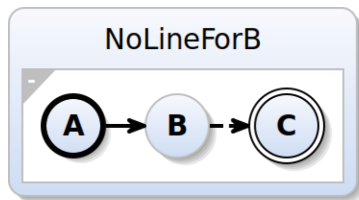
Note that when a `StateBreakpoint` is hit, said breakpoint contains the information what state it belongs to, thus the mechanism described here is not required. In that case, determining the state associated with the breakpoint along with its parent states becomes trivial. The procedure described here is needed when the execution is suspended for reasons other than a `StateBreakpoint` (e. g., a `TransitionBreakpoint` or the end of a debug step).

4.7.3 Executing Transition

Determining the currently executing transition works via the current instruction pointer of the running debug session, which Eclipse provides in the form of the line currently being executed. The simplest approach is to check, again using a regular expression, whether the current line contains an end-of-line comment matching the expected marker comment format for transitions and if so, to retrieve the transition using the information from the comment as in the previous cases.

Another option is to use the AST representation in the editor to determine not only the current line, but the surrounding syntactic construct (e. g., a block statement or method). This way, any line within the body of the transition can be mapped to the marker comment on the guard, thus allowing more precise highlighting. This option can be explored further in the future.

Here, it is worth noting that the executing transition should be highlighted if a user step ended on the transition or if a transition breakpoint on it was triggered. However, when a state breakpoint is triggered, the first outgoing transition should not be highlighted to avoid confusion, even if the first line of the method body (where the `StateBreakpoint` is placed) is often also the location of the first transition's guard (where the corresponding `TransitionCheckBreakpoint` is placed).



(a) A simple SCChart.

```
switch (state()) {
  case NoLineForBEntry:
    pauseB(State._L_0);
    if (true) break;

  case _L_0:
    termB();
    if (true) break;
}
```

Listing (4.5) Priority-based code generated from the SCChart. There is no line of code corresponding to state `B`.

Figure 4.11. A priority-based compilation example. Source state B does not appear in the generated code.

4.8 Implementation Approach for Priority-Based Compilation

Even though the state-based code generation approach generates particularly readable and easy-to-debug code, other compilation approaches for SCCharts exist and have been implemented. While these other approaches generate code that is less readable for humans, the generated code may be a lot more performant than with the state-based variant [SMH18]. As further examples of how model-based debugging can be realized for different code generation approaches, this section presents an implementation concept for the *priority-based approach* while the following section focuses on the original *netlist-based approach*. Both approaches were introduced in Section 2.2.

4.8.1 Marker Comments

To allow model-based debugging of priority-based code, marker comments must be introduced in appropriate places as described in Section 4.2. However, finding appropriate marker locations is more difficult than for state-based code since the priority-based code has a vastly different structure than the source model.

For SCCharts, the priority-based compilation system transforms the model into an SCG before generating code. Through tracing, each SCG node can be mapped to the original model element it was generated from. This could, for example, be used to annotate each assignment or each guard originating from a transition with a marker comment at the end of the line similar to transition marker comments in the state-based approach. However, some model elements may not appear in the SCG at all. Since the code is generated from the SCG, they do not appear in the code, either. As an example, consider Figure 4.11. While state A as the initial state appears and has an associated `pause` statement in line 3(since its outgoing transition is delayed) and state C appears, too (in the form of the `term` statement in line 7 that terminates the main thread), there is no trace of state B in the generated code. B neither causes a delay, which would require a `pause`, nor does it have guarded outgoing edges requiring a guard evaluation.

In state-based code, this problem does not occur since even an unguarded and non-delayed transition needs to execute at least one line in its body to set the region's active state to the transition's target. Since priority-based code does not actively track the current state, these transitions can simply be omitted. For languages other than SCCharts, similar problems may occur.

However, whenever a state has either an outgoing edge with a guard or a delayed outgoing edge, there must be a label corresponding to it in the generated code. In either case, there is a chance that the state is not left in the same tick it is entered in, thus requiring a `pause`. When an unguarded

```
case UnguardedDelayedEntry: //State marker here
  pauseB(State._L_0); //Transition marker here
  if (true) break;

case _L_0:
  termB();
  if (true) break;
```

Listing (4.6) Generated code extract for an unguarded, but delayed transition

```
case GuardedImmediateEntry: //State marker here
  if(anInput){ // Transition marker here
    gotoB(State._L_0);
  } else {
    gotoB(State._L_1);
  }
  if (true) break;

case _L_0:
  termB();
  if (true) break;

case _L_1:
  pauseB(State._L_2);
  if (true) break;

case _L_2:
  gotoB(State.GuardedImmediateEntry);
  if (true) break;
```

Listing (4.7) Generated code extract for a state with an outgoing immediate, but guarded transition

Figure 4.12. Code examples for marker placement in priority-based code.

delayed transition goes out, the execution resumes in the next tick at another label corresponding to the next action to be executed after the state has been left. An example for such a state can be seen in Listing 4.6. In that case, placing a marker on the label of the corresponding `pause` statement will suffice. If an immediate, but guarded outgoing transition is present, like in Listing 4.7, a label marking the beginning of the state is required since the same behavior needs to resume at the beginning of each tick until the transition is taken. In the example, execution starts in line 1 in the first tick. If the outgoing transition is not taken, the execution goes to `_L_1`, where it pauses until the next tick starts. In the next tick, it resumes at `_L_2`, from where it returns to the initial label in line 1. In this cases, a marker can be placed on the line with the state's entry label.

When there are no guarded and no delayed outgoing transitions present, the marker for the state can safely be placed on the next state's entry label, resulting in multiple stacked markers on the same line. When suspending on said line due to a breakpoint, the breakpoint can contain information on what state the breakpoint was intended for. When using the stepping functionality, the intermediate state can simply be omitted and the last marker of the line (i. e., pertaining to the last state in the intermediate chain) can be assumed as active. Since there is no code associated with the intermediate state (and in particular, no priority change), there is no possibility of parallel threads performing changes in between the two states.

Similarly, markers for transitions can always be placed on the conditional statement evaluating their guard, if any, or the `pause` statement if unguarded, but delayed. For immediate, unguarded transitions, the marker can be placed on the target state. Here, it should be placed on the line *after* the label to ensure that the markers for transitions are always on executable lines of code.

4.8.2 Setting Breakpoints and Extracting Runtime Information

As described above, most states have either a label or a pause statement associated with them. The remainder are states with no inner behavior, no guarded nor delayed outgoing transitions, in which case their marker is located on the next state.

In either case, a breakpoint can be placed on the line of code after the marker, i. e., the first line of the corresponding `case`-block. If the marker stores the state it belongs to, this information can trivially be retrieved at runtime. However, if this information is lost (e. g., if the breakpoint has been recreated after a restart of Eclipse), the comment needs to be parsed as described in Section 4.7. In the rare case of multiple state markers on the same line, the breakpoint can default to the last one since the previous states do not contribute to the model's behavior and placing the breakpoint on either location is equivalent.

To place transition breakpoints, a similar logic applies. For transitions with clear associated code locations, e. g., if they have guards or effects, their marker is placed on these locations. If the transition is unguarded and delayed, the marker is placed on the `pause` statement belonging to it and if neither is the case, the marker is located on the next state's first executable line. In either case, the breakpoint can simply be placed on the line with the marker. Similarly to the state breakpoints, the breakpoint itself can store the transition information, which can be recovered from the marker comment if lost.

Information on the active states of each region can be retrieved from the program counter and the labels it stores. For the currently running thread, it suffices to mark the executing state(s) as active following the method above. All other threads are either not alive (i. e., not yet started or already terminated) or suspended on a label, which can be accessed through the program counter field. Since the labels are enum values, they can be augmented with marker strings similarly to the `activeState` values in the state-based approach. With these marker strings, the state's marker string and thus the original model element itself can be extracted and highlighted.

4.9 Implementation Approach for Netlist-Based Compilation

The netlist-based compilation approach for SCCharts was first proposed by von Hanxleden et al. in 2014 [HDM+14]. Similar approaches exist for other languages. The idea behind this approach is to synthesize a logical circuit implementing the behavior modelled by the SCChart. This circuit can then either be simulated in software by expressing it as a set of boolean variables and assignments in a language such as C or Java, or hardware can be synthesized from it via a compilation to VHDL.

While the priority-based approach has faster average tick times and scales better, the netlist-based approach has a smaller jitter in tick times and is thus more predictable [Pei17]. It is currently also the only compilation approach for SCCharts that allows for hardware synthesis.

4.9.1 Marker Comments

Similarly to the priority-based approach, a transition's effect as well as its condition can be tracked from the source model all the way down to the sequentialized SCG. For transitions with a trigger, the trigger expression is evaluated and stored in a generated guard variable. This variable is then used to determine whether the effect, if any, should be executed as well as whether the next block should be active. Placing a marker on the effect is only possible where one is present. Otherwise, the transition's trigger is only evaluated as part of the next block's entry guard, as shown in line 12 of Listing 4.8, where `anInput` is the trigger of an immediate transition with no effects. The line where the next block's entry guard is evaluated isn't a suitable place either since that guard will be evaluated regardless of whether the transition is taken, and even regardless of whether the source state is active.

However, a marker on the guard can be useful for placing a `TransitionCheckBreakpoint` anyway, as discussed in the following section. For `TransitionTakenBreakpoints`, a marker should be placed on the guard variable's declaration so that a variable breakpoint can be placed on it at runtime. Unguarded delayed transitions, as shown in Listing 4.9, can be handled similarly since they are taken when

```
public boolean anInput;
public boolean _g1;
public boolean _g4; //State marker here
public boolean _GO;
public boolean _cg1;
public boolean _TERM; //TransitionTaken marker here
public boolean _pg1;

public void logic() {
    _g4 = _pg1;
    _g4 = _GO || _g4;
    _TERM = _g4 && anInput; //TransitionCheck marker
      here
    _g1 = _g4 && !anInput;
}
```

Listing (4.8) Generated code extract for a state with an outgoing immediate, but guarded transition

```
public boolean anInput;
public boolean _GO;
public boolean _TERM; //State marker here
public boolean _pGO; //TransitionTaken &
     TransitionCheck marker here

public void logic() {
    _TERM = _pGO;
}
```

Listing (4.9) Generated code extract for an unguarded, but delayed transition

Figure 4.13. Code examples for marker placement in netlist-based code.

the previous block's guard was present in the past tick, so they have a guard expression similar to transitions with triggers.

Immediate transitions with no trigger nor effect do not appear in the generated code, similarly as in the priority-based approach, making it impossible to place a breakpoint on them directly. However, by the same reasoning as there, they do not have effects on the runtime state of the program and thus, it is safe to place the marker on the target state instead.

For states with guarded or delayed outgoing transitions, at least one guard is required since in either case, the state may not be left in the same tick it was entered. Whenever such a guard evaluates to `true`, the corresponding state must be active. Therefore, a state marker should be placed on the guard's declaration. With a similar reasoning as above, states that have a single unguarded and immediate outgoing transition can have their marker placed on the transition's target state instead.

4.9.2 Setting Breakpoints

As described previously, all guards are evaluated in each tick regardless of whether the respective state or transition is active. Therefore, placing breakpoints on the line(s) where they are evaluated causes a suspension in each tick, which is not desired.

Instead, variable breakpoints, or *watchpoints*, can be placed on the guards themselves. A watchpoint in Java is a type of breakpoint that is coupled not to a code location, but a field or variable instead. It can be configured to suspend the execution either when the variable is accessed or exclusively on writes. As all Java breakpoints, it can also be tied to a condition, which can be any Java expression evaluating to a boolean. Thus, placing a watchpoint on a transition's target's entry guard variable that is only activated on write access, coupled with the condition that the variable's value must be true, only suspends the execution the moment the guard evaluates to true, which corresponds to the transition being taken. Therefore, such a breakpoint can be used to implement `TransitionTakenBreakpoints` for guarded or delayed transitions.

`TransitionCheckBreakpoints` should be triggered every time the SCChart is in the transition's source state and the trigger is evaluated. Since being in the source state is represented by a known guard, a watch breakpoint on the transition's target's entry guard can be used here as well. Just as for the `Transition Taken Breakpoints`, it should be only be activated when the guard is written, not

when it is read. However, its condition should not be the guard itself being `true` (i. e., the transition being taken), but rather the source state's guard being `true` (i. e., the SCChart is in the source state and the guard is evaluated). Without this additional condition, the breakpoint would trigger each tick regardless of active state.

`StateBreakpoints` can be placed similarly to `TransitionTakenBreakpoints`. If there are multiple outgoing transitions for a single state, multiple guards may be present. Placing a breakpoint on each one of them is required. To avoid multiple suspensions, `TransitionWatchBreakpoints` can be used as presented in Section 4.6 for the state-based approach.

4.9.3 Extracting Runtime Information

When a breakpoint is triggered, the currently executing model element can be determined either from the breakpoint itself storing its associated state or transition or, if that information is unavailable, from the comment on the breakpoint's line. This procedure is similar to that presented for the other two approaches, only that here, the Java editor's AST may need to be used to locate a variable's declaration to retrieve the respective end-of-line comment for the watchpoints.

Executing Model Elements

However, determining the executing state or transition in other scenarios (e. g., when a debugging step ends) may be difficult. Extracting the active states from netlist-based code is not simple, either. Figure 4.14 summarizes possible ways of determining the active and executing state for netlist-based code and their advantages and drawbacks.

If a step happens to end on a line with a marker comment, that comment can be used to determine the executing state. Otherwise, however, there is no way to determine the executing source model element just from the current code location. All code is located in a single method, the guard's names do not permit any conclusions as to the model element's original names and the SCG nodes have been reordered. Therefore, searching through the previously executed lines until a marker comment is found does not guarantee that the current line belongs to the same element.

Without greatly altering the structure of the generated code, determining the executing model element purely from a line of code may only be possible by adding marker comments to every line in the generated code, annotating the source model element it was generated from. Whether or not this method is sensible depends on different factors. On the one hand, the code becomes less readable through the introduction of a large amount of clutter and its size increases considerably. In turn, debugging the code using stepping becomes viable. The increased code size may be tolerable since comments do not influence the runtime behavior or executable size outside of debugging. One could argue that the decreased readability is less of a problem than it would be for other code generation approaches since netlist-based code is hardest to read by humans out of all three approaches anyway [SDH19]. Therefore, even without adding any clutter, the netlist-based approach can be considered a poor choice for scenarios where the generated code is intended to be read by humans.

Active States

With netlist-based code, finding the active states of an SCChart requires extensive search. There is no central variable that stores the active state for each region. Instead, each state has its own guard(s) irrespective of region. One could scan all guards belonging to states and, if `true`, consider the corresponding state active. However, since all guards are re-evaluated in every tick, this method may yield a partially incorrect list depending on the point of the tick function where it is applied.

	Comments on every lines		Scanning all guards
+	Can determine executing state reliably	+	No additional comments required
+	No influence on runtime behavior	+	No influence on runtime behavior
-	Visual clutter in the source code	-	Requires scanning all variables in every tick
-	Large increase in source code size	-	Requires large amounts of bookkeeping

Figure 4.14. Approaches to determining active and executing states in netlist-based code and their advantages and drawbacks.

If a state's guard is `true` and has not yet been evaluated in the present tick, it can only be seen that it was active in the past tick. Since the SCG sequentialization preserves the ordering of code in each thread, neither its outgoing transitions nor any of the following states have been evaluated, and the state can thus still be considered active. If the guard has been evaluated to `false` in the present tick, it can be assumed as inactive since the same guard cannot be evaluated again in the same tick due to the SSA-like variable separation. However, if the guard has been evaluated to `true` in the present tick, that only means that the state has been active at some point during the present tick. Its outgoing transitions or any following states may have been evaluated already, too, which means that it may no longer be active. In such a case, one could keep the state as potentially active. If another state from the same region is found that is potentially active in the same tick, but evaluated later, the first state should be marked as inactive. If no further states from the same region are found to be potentially active, one can consider the first state truly active.

However, this method requires a large amount of bookkeeping, mapping of states to their regions and keeping track of guards that have been evaluated already, which may introduce a major performance overhead especially when debugging larger models.

Chapter 5

Implementation

After the design decisions and goals were outlined, this chapter presents the implementation details for the demonstrator created as part of this thesis. As mentioned before, the implementation provides a fully functional model-based debugging environment for state-based SCCharts compilation to Java. Where applicable, changes required for other target languages will be discussed.

5.1 Plugin Setup

The demonstrator has been integrated into KIELER, which is based on the Eclipse platform. Since Eclipse is highly modular, all code for the demonstrator needs to be organized into plugins. Since its functionality is specific to SCCharts and part of the core development process, there is no separate debugging plugin; instead, all code from this project has been integrated into the `de.cau.cs.kieler.sccharts` and `de.cau.cs.kieler.sccharts.ui` plugins (from now on referred to as "the SCCharts plugin" and "the UI plugin") already containing all previous core functionality.

The separation into UI and non-UI plugins serves to ensure that all code with UI dependencies (e. g., views, menu contributions, breakpoint listeners) is collected in a joint plugin while UI-independent components (e. g., parsers, compilation systems) reside within a plugin that has no such dependencies, thus allowing easier use of these components outside the IDE where the UI dependencies would be unavailable.

5.2 Modularity

This section gives an overview of the different components used in this project. To make it as easy as possible to adapt the tool for different host languages and compilation approaches, the code has been split to allow the reuse of code wherever possible. Figure 5.1 presents the different components and whether they can be reused in scenarios other the one implemented here. Details on the implementation of each component is given in the following sections.

As can be seen in the topmost group, several components are completely independent from both the host language Java and the state-based compilation approach being used. This includes the `DebugAnnotations` processor, which is responsible for generating the marker comments from the tracing information provided by the compiler environment as described in Section 4.5.1. Since this operation runs on the model level, it is independent from the further compilation backend as long as said backend supports including the marker annotations in the generated code appropriately. The group also includes all code related to displaying, highlighting and interacting with the diagrams, since it was designed to operate purely on model elements, thus being independent from all specifics of host language and generation approach.

The center group contains components that are specific to Java and the Eclipse JDT, but can still be reused for other compilation approaches generating Java code. This group includes the debug listener

Component	Location	reusable across host languages	reusable across compilation approaches
DebugAnnotations processor	processors.statebased	✓	✓
Diagram Actions & Hooks	ui.debug.actions / ui.debug.hooks	✓	✓
Highlighters	ui.debug.highlighting	✓	✓
Debug Diagram View	ui.debug.view	✓	✓
Debug Listener	ui.debug.breakpoints	✗	✓
Breakpoint implementations	ui.debug.breakpoints	✗	✓
Diagram Part Listener	ui.debug.view	✗	✓
Template adaptations	processors.statebased.lean.codegen.java	✗	✗
Debug Breakpoint Manager	ui.debug.breakpoints	✗	✗
Model Breakpoint Manager	ui.debug.breakpoints	✗	✗

Figure 5.1. Different components and their reusability across different languages and compilation approaches. Note: prefix `de.cau.cs.kieler.sccharts.` omitted on all locations.

that listens for Java breakpoints being triggered, as well as debug events emitted when the debug run starts, stops or is interrupted. Furthermore, it contains concrete implementations of the different breakpoint types for Java, which are trivial and thus not further described here, and a part listener used to detect changes of the active editor to ensure that the diagram view always displays a diagram matching the code currently being viewed. The latter is Java-specific since it extracts the model path from Java editors when opening, which would need to be adapted to work with editors for other languages.

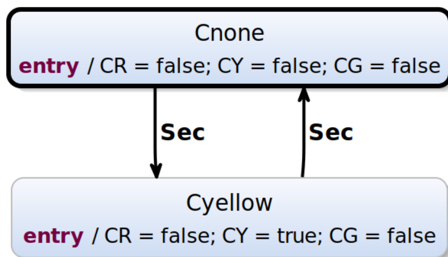
The final group includes adaptations to the code generation template to include the additional marker comments, which is only applicable to this particular scenario and needs adaptation for any other use case. It also includes the main debug breakpoint manager responsible for retrieving the runtime information from the current execution, as well as the model breakpoint manager responsible for managing breakpoints and highlightings for a specific model, both of which are highly specific to both the compilation approach and the Java language.

5.3 Changes to the Compilation Chain

As described in Section 4.5, the compilation chain needs to be adapted in order to preserve the tracing information until it is needed by the tool during debugging. For this purpose, marker comments are attached to the transformed model elements, which are then included in the generated code by the code generation step.

The marker comments include a name hash rather than the object's `hashCode` provided by Java. This way, each state and transition can be uniquely identified even if the model has been reloaded (and the Java objects representing them are thus no longer identical). In Java, hashes over strings do not depend on the string's object identity as they would for other objects, but only on the characters included in them. Therefore, they are a prime candidate for this use case. In other languages, one would need to ensure that the hashing algorithm used has the same property.

The hash is computed over a fully qualified name of the model element since multiple states may have the same names if they are located within separate regions. Therefore, the `DebugAnnotations`



(a) Extract fron source model under compilation. The code on the right is generated from a state introduced through an `abort` transformation on a higher hierarchy. level.

```
/**
 * State Cnone (-76957626)
 */
private void TRAFFIC_LIGHT_state__EA_Exit8
  (TRAFFIC_LIGHT_regionCarContext context) {
  if (context.delayedEnabled && (!iface.Error)) {  // Transition
      Error (Priority 1) -> Normal (-450882192)
    context.delayedEnabled = false;
    context.activeState = TRAFFIC_LIGHT_regionCarStates.
     _AABORTED3;
  }
  else if (context.delayedEnabled && (iface.Sec)) { //
     Transition Cnone (Priority 1) -> Cyellow (1365165695)
    context.delayedEnabled = false;
    context.activeState = TRAFFIC_LIGHT_regionCarStates.
     CYELLOW13;
  } else {
    context.threadStatus = ThreadStatus.READY;
  }
}
```

(b) Java code generated from state `Cnone`. Names shortened for better readability.

Figure 5.2. Source model excerpt and generated code.

processor concatenates all names of parent states and regions, which are available from the source model, before computing the hash and adding it to the marker comment. At runtime, the same (static) method can be called to compare the hashes found in the generated code to those computed from the source model, thus recognizing the model elements. An example of a source model excerpt and a part of the code generated from it with the marker comments can be seen in Figure 5.2.

For states, the comment format is `State <stateName> (<nameHash>)`, where `<stateName>` is the name of the source model state and `<nameHash>` is the fully qualified name hash discussed above. Transitions cannot be uniquely identified using only the source and target states since there may be multiple transitions connecting the same pair of states. Therefore, both the comment format and the hash computation for transitions include not only the source and target states, but also the transition priority in the source state. The full comment has the format `Transition <sourceStateName> (Priority <priority>) -> <TargetStateName> (<transitionHash>)`. Here, the `<transitionHash>` contains the fully qualified names of both associated states as well as the transition priority.

Before this thesis, the state-based compilation approach for Java did not include any comment annotations in the generated code, even if they were present in the source model, e. g., due to the user attaching them to a model element in the SCCharts editor. Now, they are included above the method as Javadoc comments if attached to a state and after the guard as end-of-line comments for transitions. If multiple comment annotations are present for a single model element, they are added in the order present in the model.

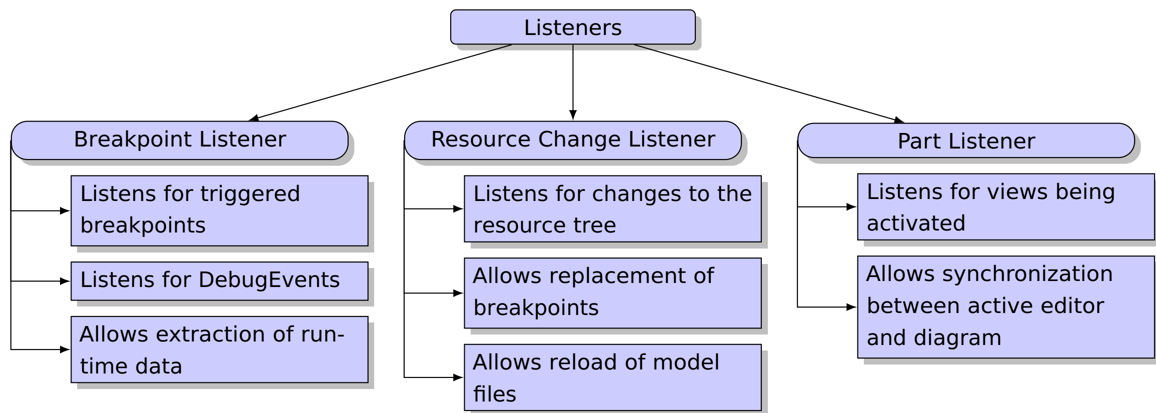


Figure 5.3. Different listeners and their uses.

5.4 Listeners

A key implementation challenge is to keep the generated code being displayed in the editor, the source model, the breakpoints and their associated model elements and the diagram view with the information on the running debug session synchronized at all times. For this purpose, a set of *listeners* is used to catch Eclipse-internal events that require updates to some of the debugging components. An overview of the listeners and their purposes can be found in Figure 5.3.

5.4.1 Breakpoint Listener

One key component is a `JavaBreakpointListener` that also listens to `DebugEvents`. It serves to notify the main `DebugBreakpointManager` of breakpoints being added and removed, debug runs starting and ending and the debug session being resumed and suspended (the latter either due to breakpoints being hit or a step ending). All of these events are categorized and filtered, then passed on to the `DebugBreakpointManager`, which then reacts appropriately.

To extend the debugging interface for other host languages, it is either possible to add a new listener propagating another set of debug events to the same breakpoint manager, extending the class to provide additional functionality to support said language, or to implement a separate debug manager fed by another listener.

5.4.2 Part Listener

To ensure that the `DebugDiagramView` is always synchronized to the currently active editor, a `PartListener` needs to be used. If registered at startup, it is notified each time a new view is activated (i. e., clicked by the user or otherwise brought to focus). The notification events are filtered and all those that concern any editor are passed on to the `DebugBreakpointManager`. With this information, the active model can then be changed either to match the newly activated generated code, if present, or to a blank placeholder model.

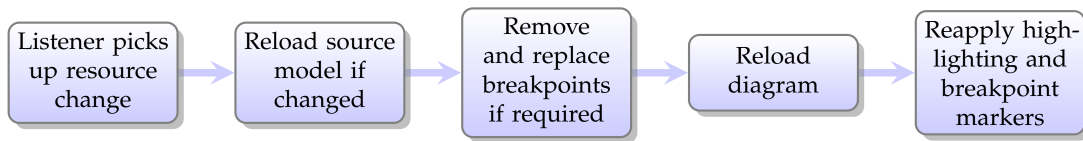


Figure 5.4. Process triggered by resource changes.

5.4.3 Resource Change Listener

The `DebugBreakpointManager` keeps a central registry of SCCharts models and their associated `ModelBreakpointManagers` and highlighters based on the file paths the models can be found under. This way, the identification can persist even after the editors, model objects and names in the source files have changed. Using this method for the model files to buffer them across editor activations improves performance since they do not need to be loaded from mass storage each time an editor is activated.

To ensure that model files are reloaded if they change on disk, a `ResourceChangeListener` is used. Without this listener, changes in the source model cannot be reflected in the diagram displayed by the debugging tool without a restart of the entire IDE. This listener implementation is short and thus realized anonymously within the `DebugBreakpointManager` class. All of the effects caused by the `ResourceChangeListener` are summarized in Figure 5.4. The different components involved in managing and highlighting models can be found in Figure 5.5.

Another scenario where this listener is required stems from the way Eclipse handles breakpoints in source files that have their content changed externally. This situation, while not as prominent during manual testing, is more common in a real-world scenario where the compilation of a changed SCCharts model is not carried out manually using the KIELER compiler selection view, but rather by an external *Continuous Integration (CI)* tool or any automatic build service such as *Maven*[1]. In case of such an external update to the file opened in an Eclipse editor, the content is refreshed, leaving all line breakpoints on the lines where they were before, even if those lines are now blank or otherwise non-sensible breakpoint locations.

Since the breakpoints here have associated model elements, it is especially crucial that they are always located in the right locations. Therefore, the `ResourceChangeListener` detects changes to the workspace's resource tree, both from within Eclipse and external, and notifies the `DebugBreakpointManager` about them. If a generated code file with breakpoints in it is affected, all breakpoints in the respective file are cleared and replaced in the updated positions. If a breakpoint's associated model element is no longer present in the model, the breakpoint is removed.

[1] http://maven.apache.org

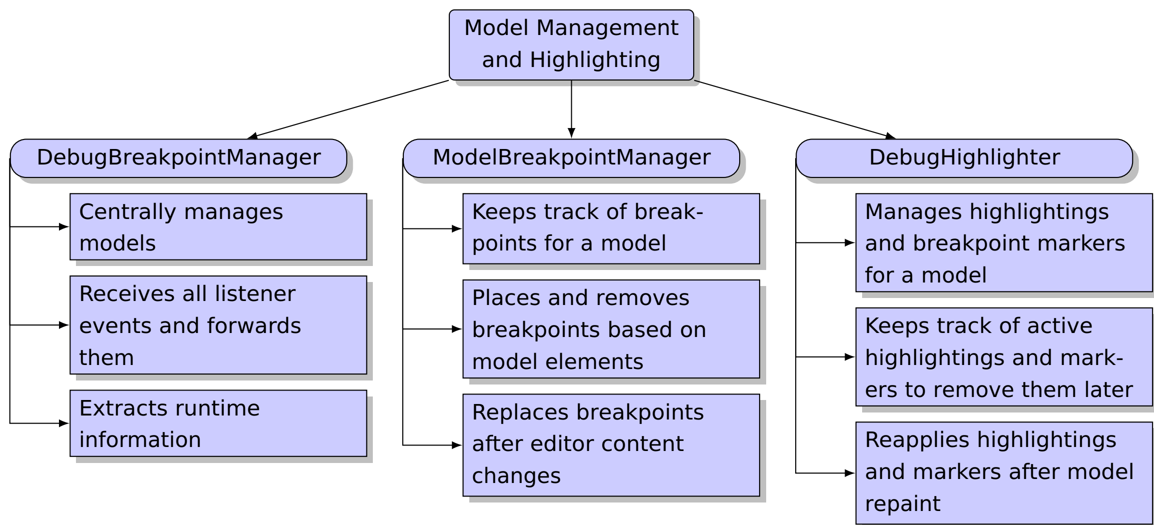


Figure 5.5. Model management and highlighting components.

5.5 Diagram View and Highlighting

As mentioned in Section 4.4.1, the `DebugDiagramView` used to display the source model along with highlightings is derived from KIELER's KLighD view. Since almost all required functionality is present in KLighD already, the implementation of the view itself contains mostly wrappers for KLighD functions to ensure easy and safe use elsewhere in the project. Most of the differences to the regular KLighD view in terms of highlighting and editor linking are caused by the listeners and other components using the view differently, not the view itself.

5.5.1 Highlighting

To ensure correct highlighting even if multiple models are running at the same time and the user switches between them, there is a separate `DebugHighlighter` object for each model, managed centrally by the `DebugBreakpointManager`. These components and their purposes are summarized in Figure 5.5. This highlighter object can add and remove highlightings for the different model elements on demand, using the mapping between source model and displayed diagram provided by KLighD. Depending on whether active or executing elements are highlighted, new backgrounds and foregrounds are dynamically added to and removed from the diagram. The `DebugHighlighter` is also responsible for adding and removing breakpoint markers to the diagram on request. This can be achieved by creating a new `KEllipse` with the appropriate size and color, then adding it as a child to the desired graph element.

The Highlighter also offers an option to clear all highlightings, which is used each time the execution resumes or terminates so that outdated information is no longer displayed. In case highlightings are lost due to a new diagram layout (e. g., after hitting the refresh button on the diagram view or after editors were switched), a `reapplyAllHighlights()` method offers an option to update the mapping between the unchanged source and the newly created graphical model, then re-highlight everything as before.

In case the source model has changed on disk and is reloaded, the mapping cannot be restored as simply since now, both the source and the graphical model are lost. In that case, the

ModelBreakpointManager responsible for the model can provide a mapping from old to new model elements, which is implemented based on the full name hash mechanism described in Section 5.3. If a highlighted model element is still present and none of its parents have been renamed, it can be recovered and re-highlighted using this method. If the element cannot be found, the highlighting is deleted. As soon as the generated code is updated and a new step is performed, the highlighting works correctly again.

5.5.2 Synthesis Hooks

To allow the user to interact with the diagram, KLighD provides an interface for Actions that can be attached to graphical elements during the diagram synthesis and then triggered by the user, e. g., by double-clicking the model element, depending on the action.

To ensure that the user can set breakpoints on model elements, SynthesisHooks are registered via the Eclipse extension point mechanism to add appropriate Actions to all model elements where the user should be able to trigger breakpoints. These hooks are executed along with the regular layout run each time the diagram view changes its content.

When an Action is triggered, KLighD passes an ActionContext object to it, which then allows it to determine what model elements have been clicked, which is reported to the DebugBreakpointManager.

5.6 Extracting Runtime Information

As described in Section 4.7, runtime information can be extracted from generated code. Whenever a breakpoint is triggered, breakpoint listeners are notified and given both the breakpoint responsible for the suspension and the Thread object that has been suspended. The Thread object includes all runtime variables, stack frames and other information required for determining active and executing model elements. On the other hand, the breakpoint includes information on the model element associated with it and the type of breakpoint, influencing what needs to be highlighted.

Visualizing the runtime information in the diagram view requires the correct model to be present. The diagram view is locked to the active editor, and Eclipse automatically switches to the editor associated with a breakpoint when it is hit. However, breakpoint managers are notified before the editor is switched. Thus, if the breakpoint is located in the editor that was active before, the correct model is present in the diagram view and correctly registered. If the breakpoint causes an editor switch, runtime information cannot be visualized straight away and the marker comments are unavailable at the time where the breakpoint is triggered. It then has to be registered that a certain breakpoint was hit, and when the editor switch is triggered by Eclipse and the PartListener induces a model switch, the runtime information can be extracted and visualized.

This delay between the breakpoint being triggered and the editor becoming active is also the reason why the path to the source model has to be included as a variable rather than a comment, as mentioned in Section 4.2. When a breakpoint is hit, the tool must check whether the suspended runtime thread matches the currently loaded model and the active editor. Since it cannot rely on the editor content and thus comments to do that, the runtime thread's memory must contain a model identifier. For this purpose, the model's source file path is used.

Chapter 6

Evaluation

To evaluate the benefits of model-based debugging, the demonstrator implementation has been given to a group of around ten professional developers working on safety-critical railway systems using C, Java and SCCharts. All of them were familiar with SCCharts and Java debugging before and therefore able to evaluate the benefits of a new tool compared to their usual workflow.

A demonstration of the tool and the workflow it proposes has been given to them. Since this demonstration took place several days before they started actually working with the tool, an additional *Visual Debugging Cheatsheet* has been handed out to remind them of the available functionality and how to access it. The cheatsheet can be found in Appendix A. After a week of working with the tool, a questionnaire was handed out to them to evaluate three main questions:

- How well designed is the UI? In particular, how intuitively does it integrate into the existing workflow?
- How stable and performant is the tool? In particular, does it run slowly? Do crashes / exceptions occur?
- How useful are the provided features? Is there any key functionality missing?

The full questionnaire can be found in Appendix B. Out of all candidates, two completed the survey[1]. Despite the low number of participants, their expertise makes the results from said survey valuable. An evaluation of the results from this study can be found in Section 6.1.

Nevertheless, a second study has been conducted to reach a larger number of participants. For that study, both advanced master's and PhD students of computer science have been given an intentionally errorneous model and a set of bug reports with the assignment to find the line of code causing the bug and describe how to fix it. Since some of the participants here were less proficient both in debugging Java code and SCCharts development than the professionals participating in the first study, two groups were formed. One group used the model-based debugging demonstrator while the control group used regular Java debugging and the standard SCCharts simulation without any new debugging features. The study was conducted completely online and participants were required to run a development version of KIELER on their personal computers[2]. While a short timeframe and some technical issues prevented some candidates from participating, five volunteers completed the second study. Details on the settings, the used model and an evaluation of the results can be found in Section 6.2.

Finally, Section 6.3 relates the results from both studies and draws conclusions for the model-based debugging approach in general and the demonstrator implementation in particular.

---

[1]The low percentage of participants completing the survey in time may be partially caused by the COVID-19 pandemic and the logistical challenges imposed by it.

[2]A more controlled setting with prepared computers and a personal introduction were not achievable due to the continued COVID-19 regulations

6.1 Evaluation with Professional Developers

At the beginning of the questionnaire, the participants were asked to give information on the models they worked with and how long they used the tool before filling the survey. Both participants stated that their models had just under 40 states and they used the tool for 2 and 5 hours, respectively. While this is longer than the fifteen minutes given to the participants in the second study, it still is not long enough to fully get used to the tool. However, it is enough to get a good impression and to encounter obvious issues if there are any. The models both participants worked with contained concurrency and a number of `entry`, `exit` and `during` actions, but few other, more advanced SCCharts features.

6.1.1 Integration into the Existing Workflow

Three questions from the first section of the questionnaire are aimed towards the integration into the developer's existing workflow. Figure 6.1 is a *box plot*. Box plots show the range between minimum and maximum value for a dataset with a line in between marking the median of all answers. For example, to the statement "I had to change my habits to use the new tool", one developer agreed while the other disagreed, so the median of both lies in between.

As this figure shows, the participants state that the newly introduced components integrate well with the existing KIELER environment, but that they still need time to get used to the new tool. It can be expected that after some more time working with the tool, the initial issues of remembering the features will be mitigated.

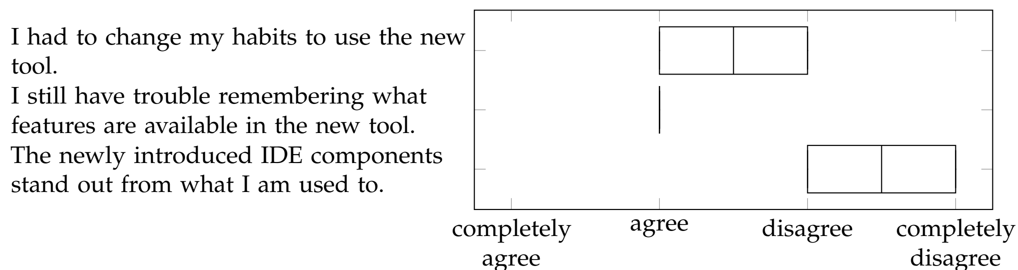


Figure 6.1. Results concerning the integration into the existing workflow.

6.1.2 UI and Useability

The remaining questions in the first section examine the perceived useability of the tool and the intuitiveness of the UI. In Figure 6.2, it is clearly visible that both participants agreed that the demonstrator is an improvement compared to using a regular Java debugger to debug generated code and that they would like to keep using the tool in the future. They also agreed that it did not take too long to get used to the tool, even though there may still be room for improvement as stated in the previous set of questions. Both agreed that the highlighting used in the diagram made sense, possibly confirming that relying on pre-existing color schemes as described in Section 4.4.3 improved diagram readability for experienced users. In terms of available features, both participants were satisfied even though one sees room for improvement.

As remarks for the first section, both had suggestions for improvement. One pointed out that contrary to the regular KLighD view known from SCCharts development, the `DebugDiagramView` does not offer any layout options, which makes it hard to navigate the diagram for users that usually rely on

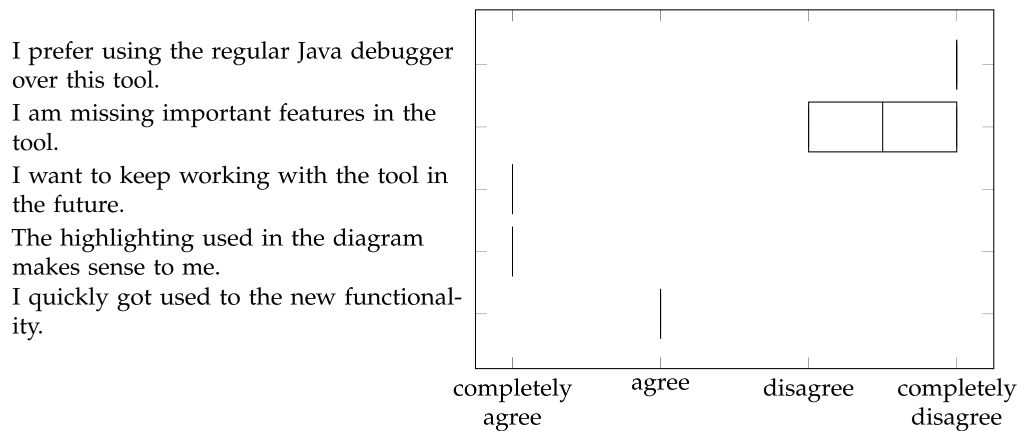


Figure 6.2. Results concerning UI and useability.

them. The other participant suggested that there should be an option to lock the DebugDiagramView so that one can switch editors, e. g., to view the SCChart's source code, without the diagram disappearing.

6.1.3 Stability and Performance

A section on stability and performance was included for two main reasons. Firstly, even though the goal of the study is not only to evaluate the demonstrator, but also the concepts presented in this thesis, collecting feedback to improve the tool is still important. Secondly, finding out whether participants had technical issues with either performance problems or crashes helps interpret their feedback on useability.

As Figure 6.3 shows, both participants largely agreed that the performance and reaction speed of the tool was adequate. However, one should consider that the models they used were rather small and thus, these results were to be expected. While one user seemed to face no technical issues at all, the other one reported that there were error messages disrupting their workflow and consequently also agreed that the tool needs to be improved before it can be released. In the remarks section, they pointed out that the tool worked quite well after some initial technical difficulties. However, they also stated that the link between diagram view and editor seems “quite fragile”, which may need improvement in the future.

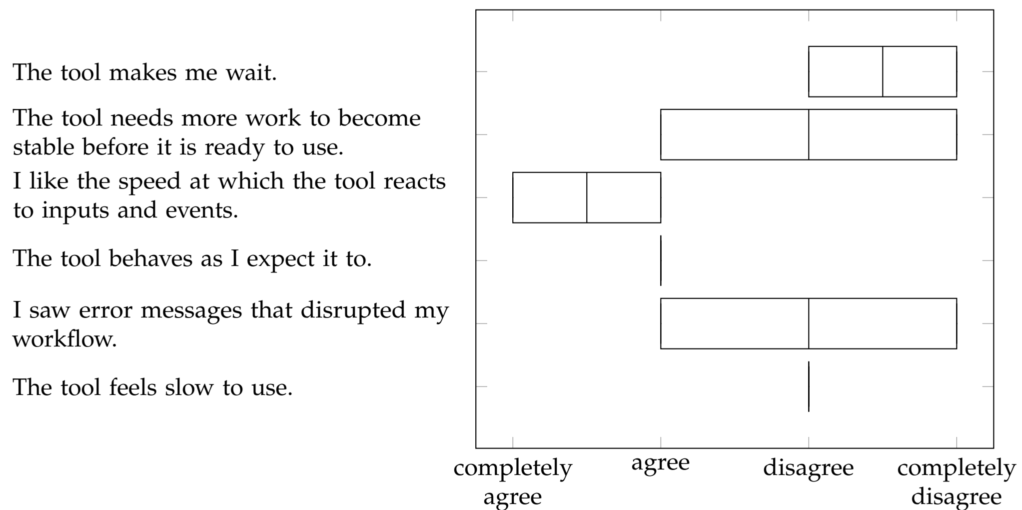


Figure 6.3. Results concerning stability and performance.

6.1.4 Questions on Breakpoints

The last section of the questionnaire contained a variety of questions on features that can be split into two main categories: questions concerning breakpoints and stepping. The first question serves to get a basic impression of whether the participants like the features currently available in the tool, to which both of them strongly agreed.

For breakpoints, the questions are designed to find out whether the participants used certain types of breakpoints currently available to determine whether some may be superfluous. At the same time, some aim towards currently unimplemented types of breakpoints and whether the participants would find them useful. Figure 6.4 shows that both participants agree that all present types of breakpoints are useful and behave as expected. From Figure 6.5, it becomes clear that both participants think that introducing more breakpoints is not required to properly use the tool and that the complexity would increase too much. Still, one of the participants states that they wanted to place breakpoints on model elements where breakpoints are not supported from time to time. This may be due to them not being used to the tool as much and may decrease over time.

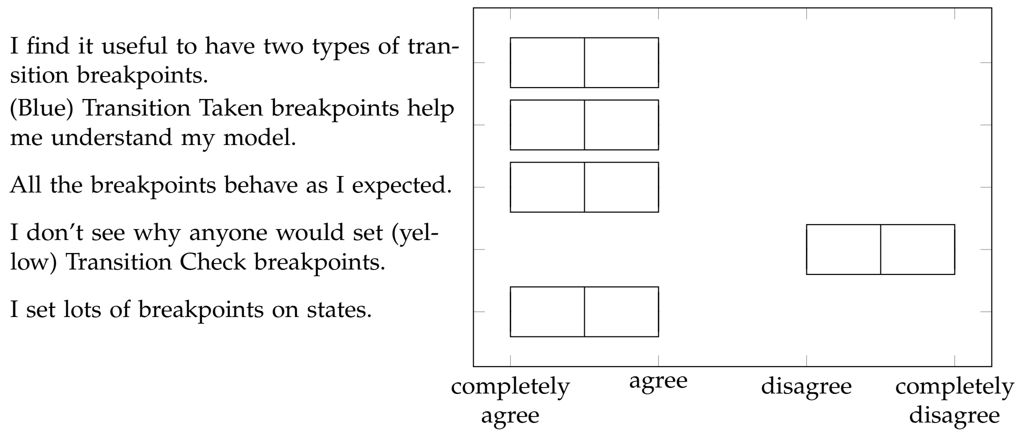


Figure 6.4. Results concerning the usage of present breakpoint types.

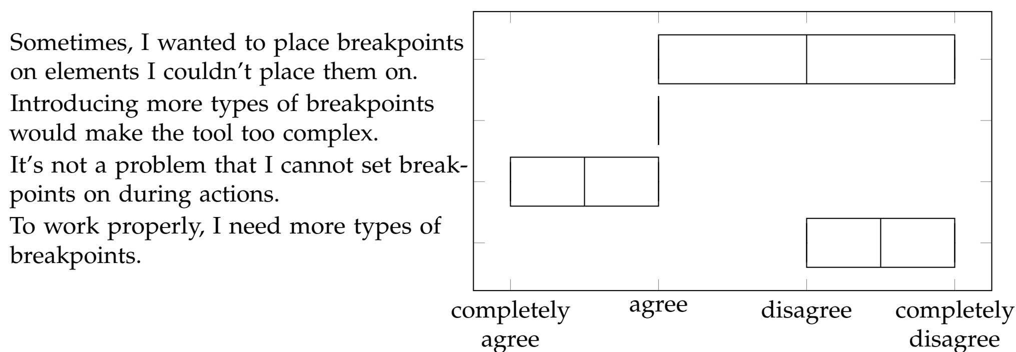


Figure 6.5. Results concerning the introduction of new breakpoint types.

6.1.5 Questions on Stepping

The final questions of the last section dealt with the stepping support provided by the debugger. Here, the opinions of the professional developers differ: One used the stepping support more and thinks that an additional model-level stepping mode on macro step granularity would be useful, while the other used stepping less and does not see the need for such a mode. However, both of them agreed that the stepping support is in a good place as it currently stands.

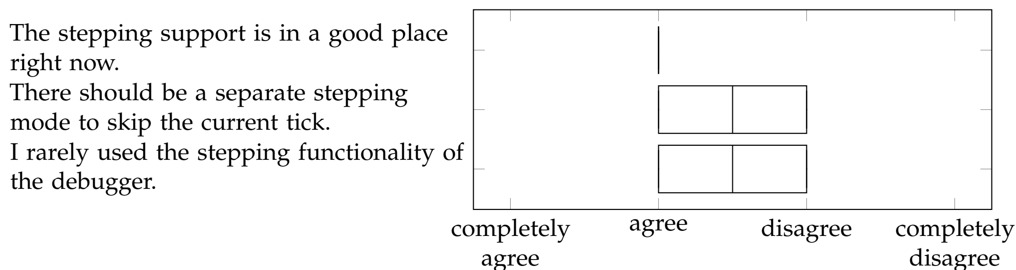


Figure 6.6. Results concerning the stepping support of the debugger.

6.2 Evaluation with University Members

The second study has been conducted using Google Forms[3]. Screenshots of each page of the survey can be found in Appendix C. Since a variety of students and university employees participated and two groups were formed, it was not possible to give an introduction in person. Instead, the online form included instructions for downloading and installing the current KIELER nightly build including the latest version of the model-based debugging demonstrator. A download link for the code to be debugged was also provided, along with instructions for importing it in KIELER. Participants were advised to not open the code before being told to do so to ensure that every participant would have the same amount of time to understand the code.

6.2.1 Study Setup

After finishing these preparations, participants were randomly assigned a group. The remainder of the survey was separate for each group. Since Google Forms does not directly support random group assignments, a question with two possible answers in randomized order was used. Participants were asked to select the topmost option, which then redirected them to one of two followup sections.

After the group assignment, each group was shown an introductory video consisting of a common welcome message and a common explanation of the study setup, the model to be debugged and the assignment as well as an introduction to the technologies to be used. The experimental group received an introduction to the model-based debugging demonstrator, its functionalities and how to use it. A link to the debugging cheatsheet already used in the first study was given to them as well. The control group instead received an introduction to both debugging the generated Java code as well as KIELER's simulation mechanism to allow them to use all of the debugging options existing prior to this thesis.

Study Assignment

Both groups were given the same assignment and the same model to debug. After the code was briefly explained to them to ensure that its semantics was clear, a list of three bug reports were given, describing problems with the model. These bug reports were intentionally unconcise to simulate a real situation where a person may report the bug who is unfamiliar with the exact structure of the model, thus not naming the responsible component, but rather describing the observable behavior. The exact bug reports and the causes for them are described in Section 6.2.2. Using the tools provided for the respective group, each participant had *15 minutes* to fulfill the following assignments:

- ▻ read the bug reports,
- ▻ open the provided code, and
- ▻ debug the code until they can locate the line of code causing each bug.

After the timer ran out or all bugs had been found, participants were asked to report for each of the three bugs

- ▻ whether they found the bug's cause,
- ▻ if yes, what file they found the cause in (source model or Java environment),
- ▻ what line the cause was located on, and

[3] https://www.google.com/forms/about/

▷ a brief description on how to fix the bug.

The last three points only serve to validate whether the participant actually identified the cause of the bug or just believed that they did. Participants were also asked to report the amount of time they used to find the bugs, which serves to determine what group was faster to find the bugs. An evaluation of this section can be found in Section 6.2.2. After these quantitative results, they were also asked to give qualitative feedback. The qualitative section differed between groups and is evaluated in Section 6.2.3.

Provided Code

Both groups were given the same code, consisting of an SCCharts model of a fictional building's alarm system, the code generated from it using the state-based approach for Java, and a matching Java environment containing necessary hostcode functions, a `main` function to instantiate the model and repeatedly call the `tick` function. The full hostcode environment and the SCCharts source file can be found in Appendix C.

The provided `AlarmSystem` SCChart can be seen in Figure 6.7. Its inputs include a `second` signal set to `present` once a second to provide a notion of time to the model as well as a `fireOut` signal. `fireOut` is used to determine when a fire has been truly put out by firefighters, since the fact that no smoke detectors are triggered anymore does not necessarily mean that it is safe to turn off the fire system. The system also has a `day` input, which switches between two operating modes, `day` and `night`.

The fictional building has a total of five rooms, each equipped with a fire sensor, a motion detector, a light and a sprinkler device. All of these are organized in boolean arrays, with each room having a unique index across all arrays. `fireSensors` and `motionDetectors` are inputs while `lightsOn` and `engageSprinklers` are outputs, supposedly connected to actuators in the building. In case of either an intruder or a fire, there is also a `alarmSound`, which can be controlled by the `AlarmSystem` using a boolean output.

The following is the intended behavior of the model:

switch lights During the day, lights turn on in a room whenever the respective motion detector is triggered. They stay on until the motion detector is no longer triggered for five consecutive seconds. When the system enters `Night` mode, all lights are disabled until `Day` mode is entered again.

detect fires When any `fireSensor` is triggered, the system enters `Fire` mode. The respective sprinkler is engaged and firefighters are alerted using the hostcode method `callFirefighters()`. The inhouse `alarmSound` is enabled and is toggled every second to achieve a beeping sound. Even if no smoke is detected any longer, all active sprinklers and the alarm sound stay on until `fireOut` is reported using the appropriate input.

detect intruders When in `Night` mode, motion detectors are used to detect intruders. When a `motionDetector` is triggered at night, the inhouse `alarmSound` is enabled and toggled every second. The system also alerts the police using the `callPolice()` hostcode method. To disable the alarm, the system is switched to `Day` mode since this allows searching for the intruder using the building's lights and prevents the police from triggering the alarm again.

6.2.2 Quantitative Results

Both groups were asked to find a set of three bugs. The following paragraphs list each bug along with the description given to the participants (in bold text), an explanation of where in the model its cause is located and the survey results from both groups.

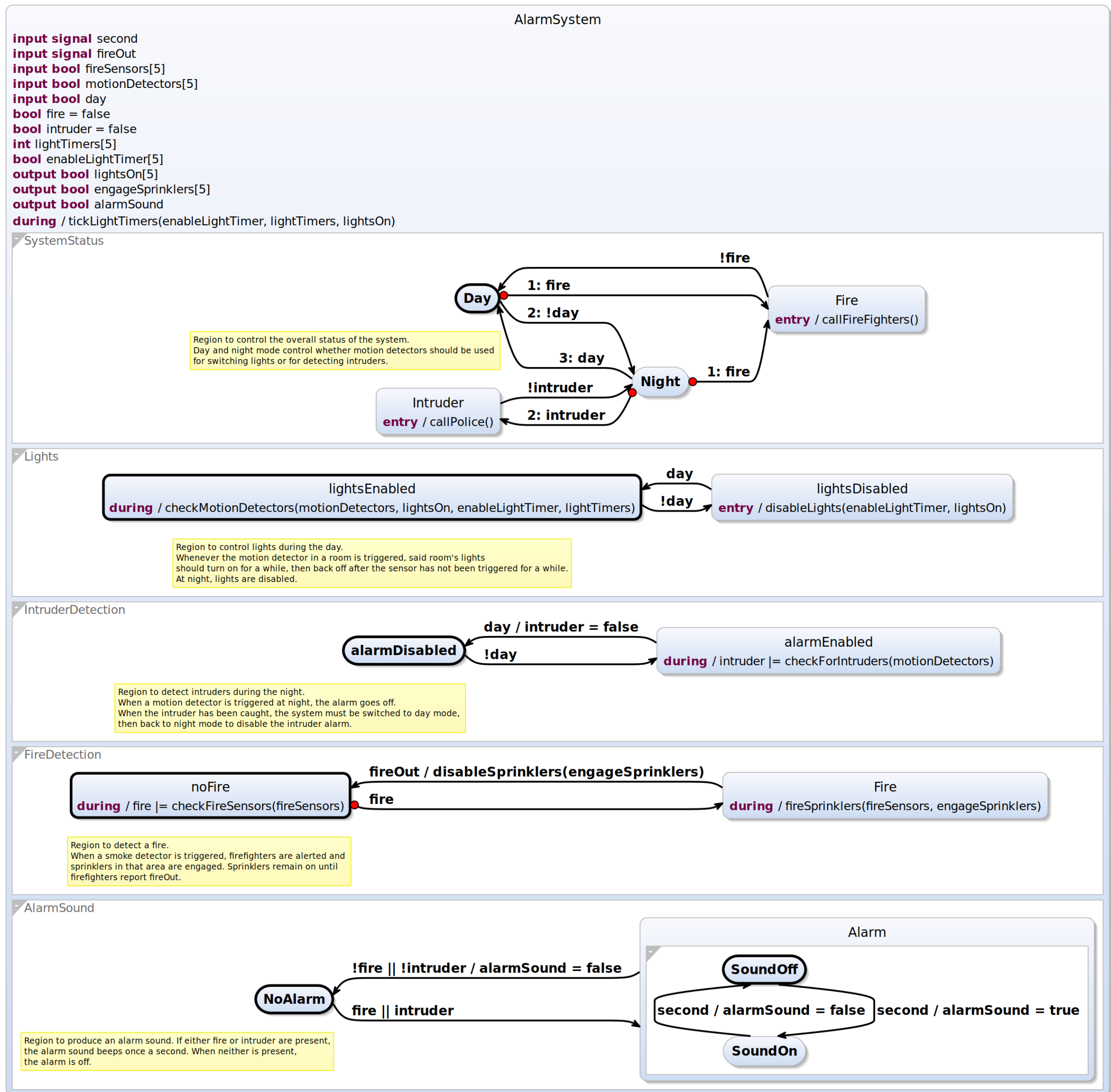


Figure 6.7. `AlarmSystem` SCChart used in the online study.

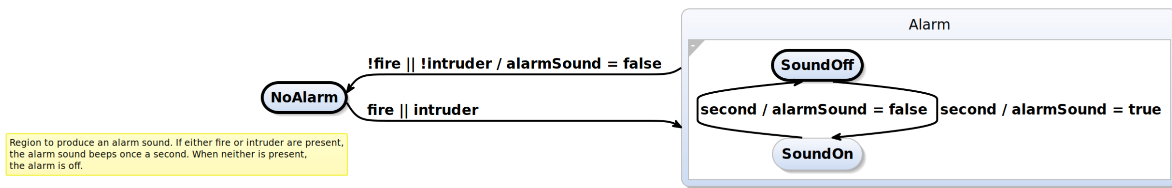
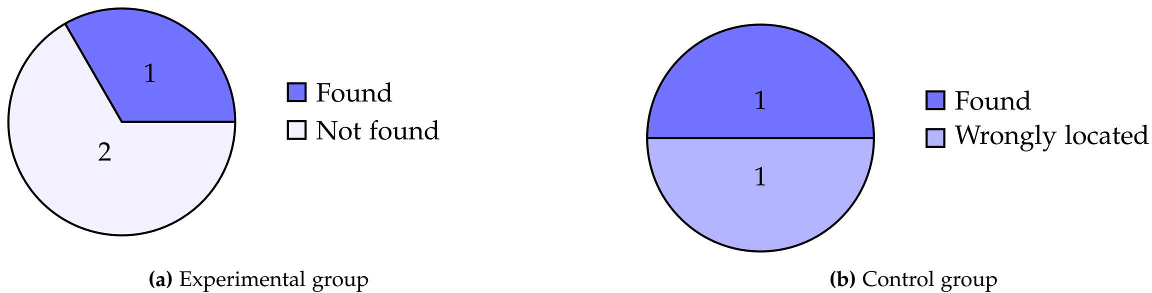


Figure 6.8. `AlarmSound` region of the `AlarmSystem` SCChart. An incorrectly guarded transition from `Alarm` to `NoAlarm` causes the alarm sound to be turned off even if one of two alarm causes is still present.



(a) Experimental group **(b)** Control group

Figure 6.9. Number of participants from either group that found bug #1.

Bug #1: No Alarm Sound

When an intruder or a fire is detected, the automatic alerts to police and firefighters work as expected, however the in-house alarm siren does not seem to operate at all.
The cause for this issue lies in the `AlarmSound` region of the `AlarmSystem` SCChart, which is shown in more detail in Figure 6.8. Instead of disabling the alarm sound when neither an intruder nor a fire are present, the state `Alarm` is left as soon as one of them is absent, meaning that a fire without an intruder (or vice versa) will cause the alarm sound to be disabled one tick after it was enabled.

This bug is not trivial to find since the `alarmSound` output is in fact correctly set to `true` in the tick after entering `Alarm`, provided that `second` is currently present. This behavior lead some participants to believe that the model did in fact work as intended. However, the outgoing weak abort transition triggers in the same tick and sets `alarmSound` back to false immediately, meaning that for an observer outside the model, the alarmSound does not turn on.

As Figure 6.9 shows, only one participant from each group found the transition's guard as the reason. One participant in the control group, even though unsure, suspected that the behavior was caused by fire sensors not being reset. However, since they are an input parameter, the model should not reset them themselves.

Bug #2: Wrong Mode after Fire

When a fire alarm has been triggered and then reset, the system does not appear to return to the correct operation mode sometimes.

This bug report is relatively vague and actually covers two bugs, however, one of them can only be found by either looking at the code or setting internal variables, not through the program's interface.

Looking at the `SystemStatus` region shown in Figure 6.10, one can see that after a fire occurred, the system always returns to `Day` mode, even if it was in `Night` mode before the fire occurred. Instead, it

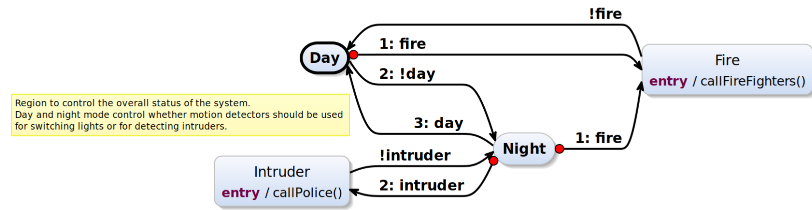
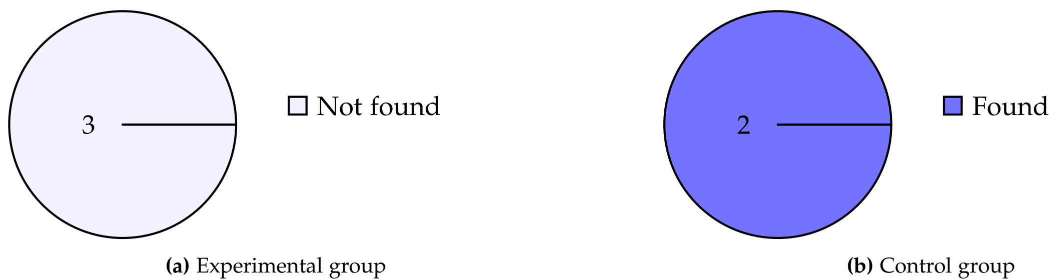


Figure 6.10. `SystemStatus` region of the `AlarmSystem` SCChart. From state `Fire`, the status always returns to `Day` mode regardless of whether it was day or night before the fire.



(a) Experimental group

(b) Control group

Figure 6.11. Number of participants from either group that found bug #2.

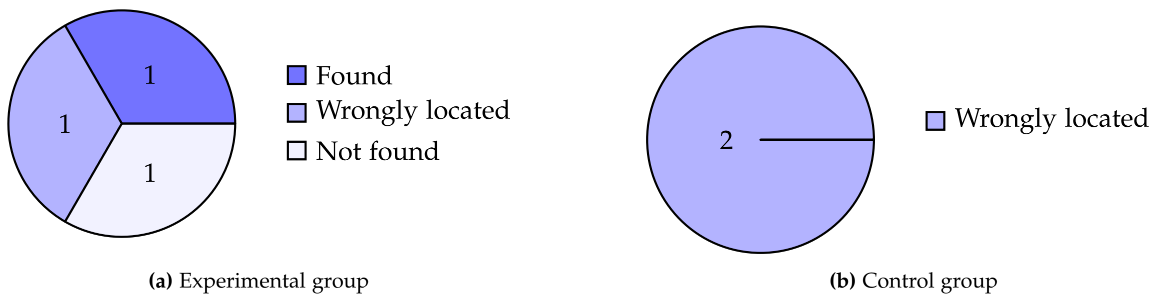
should return to either `Day` or `Night` depending on the value of the `day` input (which may have changed during the fire). This bug was reported by one participant from the control group and since it is a bug in the model, it has been counted as correctly located. However, it is not a critical bug since at night, the system status will return to the correct operation mode in the next tick.

In addition, the bug described above only occurs when setting the internal `fire` variable by hand using debugging tools. According to the model's documentation and its interface, this variable is not to be set externally. Instead, the input signal `fireOut` should be used to signal that the system may return to regular operation. Using this input, however, the system remains in `Fire` mode since all sprinklers are disabled, but the internal `fire` variable is never reset. This more critical bug has been found by the other participant in the control group while nobody in the experimental group found either bug.

Bug #3: Lights

The lights should turn on after a motion detector is triggered in the matching room and turn back off 5 seconds after the sensor no longer detects motion. The lights turn on as expected, but will then stay on during the entire day until the system changes to night mode.

As Figure 6.13 shows, all but one participant believed they found this bug, however only one of them presented a solution that would work, though it was not the intended one. Whenever a motion detector is triggered during the day, the `checkMotionDetectors()` method called by the `AlarmSystem` each tick will detect it and switch the light on. As Listing 6.1 shows, this method will set `enableLightTimer` for the respective light to `true` and set the matching `lightTimer` to 0 while also switching on the light. Consequently, `tickLightTimers()`, which is called by a `during` action on the `AlarmSystem` root state each tick and can be seen in Listing 6.2, checks whether any `lightTimer` exceeds `LIGHT_ON_TIME`. If that is the case, the respective light is turned off. From the name `tickLightTimers`, one could expect that the lightTimers are also increased every second here; however, they are not, nor anywhere else. Thus, all



(a) Experimental group

(b) Control group

Figure 6.13. Number of participants from either group that found bug #3.

light timers remain at 0 until lights are globally disabled when the system switches to `Night` mode.

The intended fix is to alter `tickLightTimers()` to also increase the light timers every second, however, none of the participants suggested this change. One member of the experimental group suggested that one should add a self transition on state `lightsEnabled` that increments light timers every second, which would be a valid option and is thus counted as correctly identified. Another participant from the experimental group said that a new region should be added to call `tickLightTimers()` regularly, perhaps overlooking the `during` action doing that already. From the control group, one participant said `disableLights()` needs to be called along with `checkMotionDetectors()`, however, that would lead to the lights being turned off immediately after the sensor stops registering movement without respecting a timer. The other member of the control group wrote that `checkMotionDetectors()` should "increment the lights in timer", which may go in the right direction, but is not sufficiently clear to be counted as correct.

```
130 /**
131  * Check whether any lights should be turned on.
132  * If so, turn on the respective lights and start a
        timer
133  * to ensure that they are turned off after an
        appropriate time.
134  */
135 public static void checkMotionDetectors(boolean[]
        mds, boolean[] lightsOn, boolean[]
        enableLightTimer, int[] lightTimers) {
136   for (int i = 0; i <= mds.length - 1; i++) {
137     if (mds[i]) {
138       lightsOn[i] = true;
139       enableLightTimer[i] = true;
140       lightTimers[i] = 0;
141     }
142   }
143 }
```

Listing (6.1) `checkMotionDetectors()` function from `AlarmSystemEnvironment.java`.

```
116 /**
117  * Check whether any lights should be turned off.
118  * This should occur if they have been on for longer
        than
119  * @link{AlarmSystemEnvironment#LIGHT_ON_TIME}
       seconds.
120  */
121 public static void tickLightTimers(boolean[] enabled
        , int[] timers, boolean[] lightsOn) {
122   for (int i = 0; i < enabled.length; i++) {
123     if (enabled[i] && timers[i] > LIGHT_ON_TIME) {
124       enabled[i] = false;
125       lightsOn[i] = false;
126     }
127   }
128 }
```

Listing (6.2) `tickLightTimers()` function from `AlarmSystemEnvironment.java`.

Figure 6.12. Extracts from `AlarmSystemEnvironment.java`. Light timers are started when a sensor is triggered and when they exceed `LIGHT_ON_TIME`, they are disabled. However, light timers are never actually increased.

Total Time

After noting the bugs they found, participants were asked to give the total time they took to find the bugs. As Figure 6.14 shows, all participants from the experimental group missed at least one of the bugs and therefore, all of them took the maximum time of 15 minutes. Both members of the control group took slightly less than that, however each one of them misjudged at least one of the bugs, so neither of them correctly found all bugs within the time limit, either.

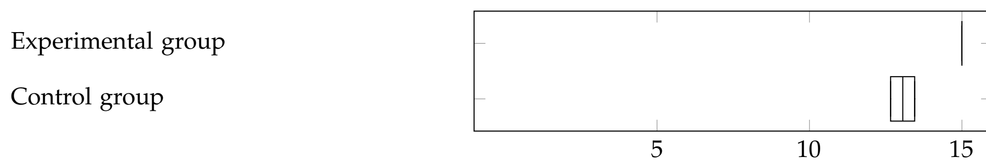


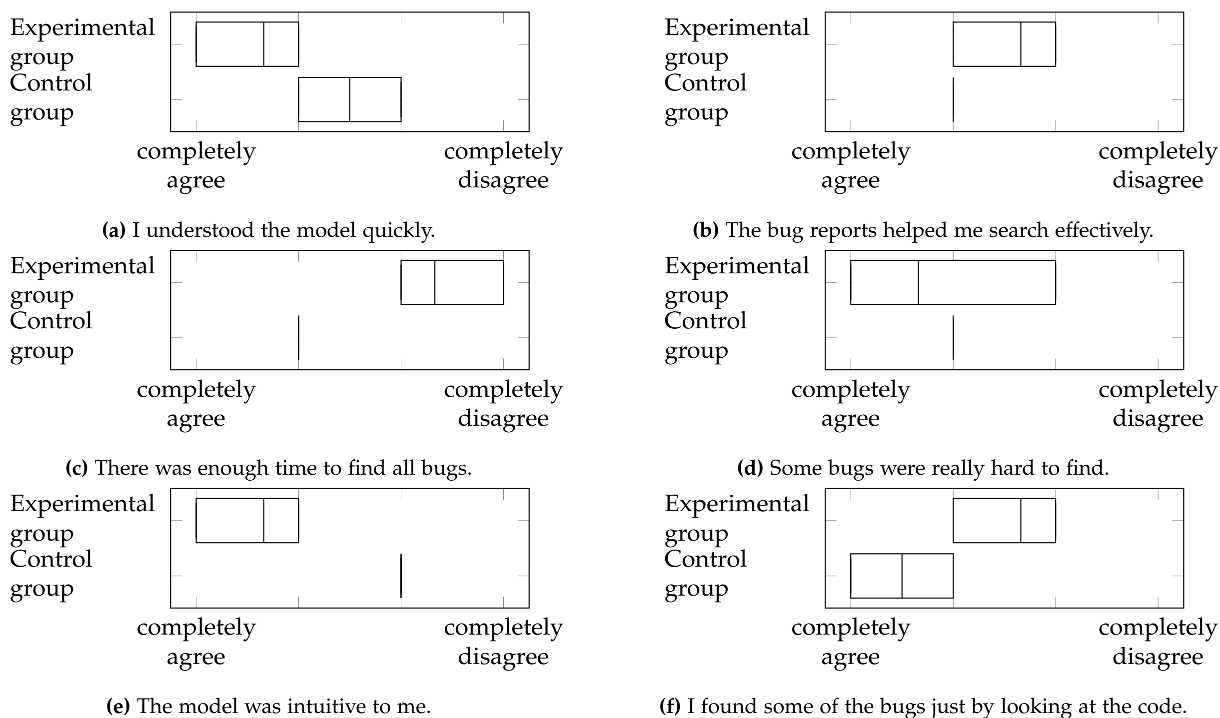
Figure 6.14. Total time in minutes taken by participants.

6.2.3 Qualitative Results

After evaluating the number of bugs found by participants, each group was presented a set of questions to evaluate their opinion on the provided model and bug reports as well as on the debugging methods they used. The first set of questions was designed to be independent of the debugging method and could therefore be identical for both groups. This allows to differentiate between perceived difficulty of finding the bugs and understanding the model to find out whether one of the methods makes it easier to understand the provided code. However, this data should be interpreted with particular care due to a large spectrum of skill levels within the participants group and its small size. The results may also be skewed by the fact that most participants were familiar with KIELER and the simulation used by the control group beforehand, making it naturally easier to use for them.

As expected, Figure 6.15 shows that the experimental group considered the bugs harder to find than the control group while also agreeing that there was too little time to find them all (Figures 6.15d and 6.15c). Figure 6.15f may show a reason for the faster times of the control group: While the experimental group states to not have found many bugs just by looking at the code, thus using the new demonstrator they were unused to, the control group seems to have spent less time debugging and more time reading the code, finding the bugs by pure experience and thinking rather than using the SCCharts simulation as intended.

The experimental group did not find the bug reports as helpful (Figure 6.15b), perhaps due to difficulties reproducing the bugs using the new debugging features. However, Figures 6.15a and 6.15e show that the experimental group found the model significantly more intuitive and easier to understand. While this may be pure chance due to the low number of participants, it may also indicate that the model-based debugger helps to understand the model under debugging. Further interpretation of the results can be found in Section 6.3.



(a) I understood the model quickly.

(b) The bug reports helped me search effectively.

(c) There was enough time to find all bugs.

(d) Some bugs were really hard to find.

(e) The model was intuitive to me.

(f) I found some of the bugs just by looking at the code.

Figure 6.15. Results of questions on model and bug reports.

Control Group Feedback

After the questions on the model and bug reports discussed above, the control group was given a small set of questions on the debugging methods they used. These mainly serve the purpose of finding out whether the users liked the debugging experience they had and whether they prefer that method over using a specialized debugger for Java code generated from SCCharts as presented in this thesis. As Figure 6.16 shows, both participants agreed that the present debugging methods of KIELER were unintuitive and not suited for solving debugging tasks like the one presented in this study. Both stated that they would rather exchange this method for a specialized debugger, confirming the need for such a tool when debugging model-based languages.

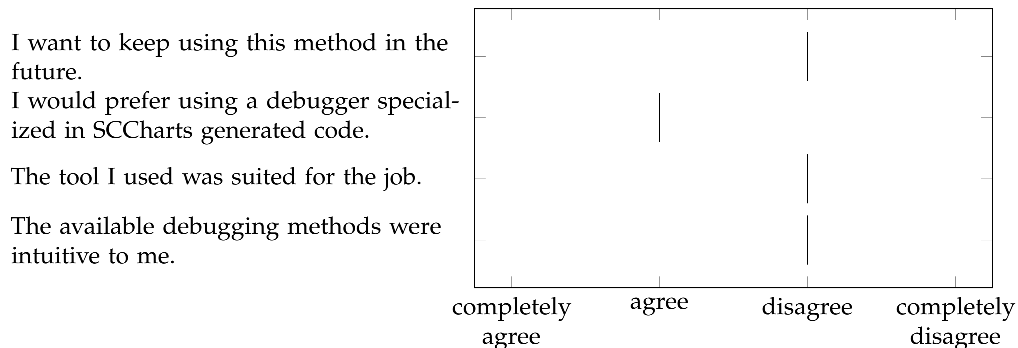


Figure 6.16. Feedback from the control group on the debugging methods they used.

Experimental Group Feedback

The experimental group was asked for feedback on the demonstrator implementation to complement the findings from the first study. To keep the total study time manageable for the participants, fewer questions were used than for the professional developers. The findings from this set of questions can be found in Figure 6.17.

The experimental group confirmed that the diagram highlighting makes sense to them, which the professional developers in the first study had established. Contrary to them, the participants of the second study stated that they had little to no trouble remembering the available features, which perhaps is due to the short time between introduction and actual use in the second study compared to the first one. The participants also agreed that they want to keep using the tool in the future and prefer it over the SCCharts simulation and a regular Java debugger.

However, a central point of criticism was that the demonstrator needs a better way of setting variable values during the execution. Currently, if one wants to change values of any variables, be it internal or interface ones, that has to be done using the Java variables view provided by Eclipse. This method requires knowledge about the memory structure of the generated code and is especially difficult if one never used it before, just like the participants.

Setting values during the execution is made even more difficult due to the different timing of reading variables. Since the SCChart is sequentialized during compilation, model elements are executed in a fixed order each tick. If one wants to set a variable during a tick, it may well be that when a certain breakpoint is reached and one can thus edit the variable, the point where it is read has already been executed for the tick. Especially for signals, this is a major problem since their value is reset after each tick, requiring the user to find a breakpoint location before the read access to the variable by themselves.

Consequently, participants rated the tool as less intuitive than the professional developers, who perhaps did not encounter this issue. Some of the participants also stated that the tool needs more features and improvements before being released, explicitly requesting a better option to set variable values during the execution. Concepts for how this can be achieved can be found in Section 7.2.

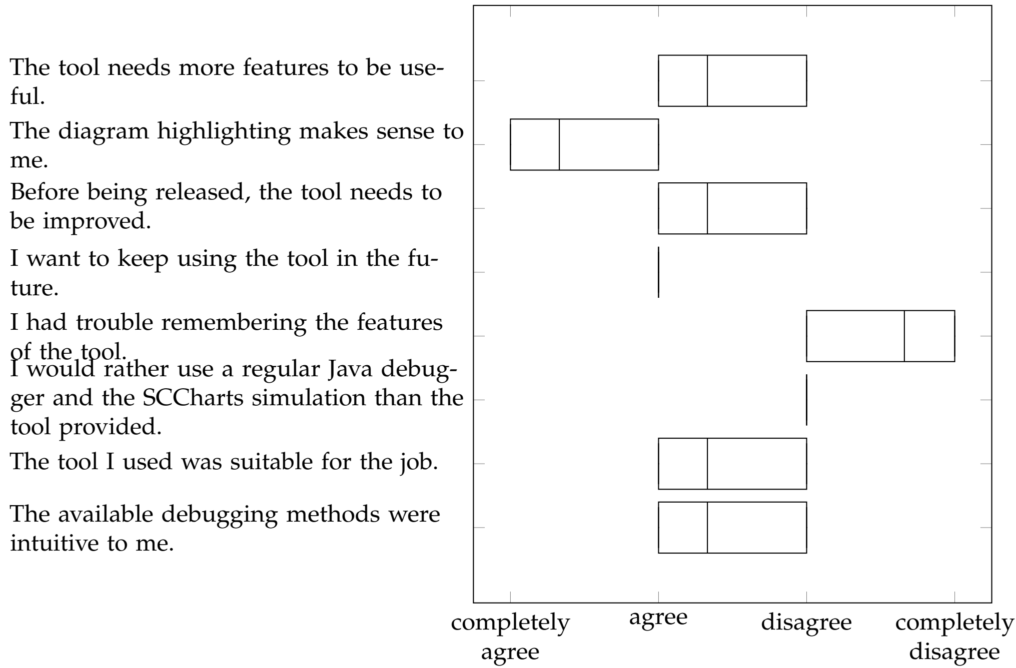


Figure 6.17. Feedback from the experimental group on the demonstrator.

6.3 Evaluation Summary

The two studies described in this section both gave mostly consistent results within each group. Even though it needs to be emphasized once again that neither study had enough participants to make the results reliable, they show a clear trend that could be confirmed or refuted by a followup study with more participants in the future.

The combination of the two studies suggests the following trends concerning the demonstrator implementation:

The demonstrator is superior to a regular debugger and the KIELER simulation. All participants agreed that using the new debugger is preferrable over the pre-existing variants. Everyone who used it wants to keep using it. The control group from the second study did not use it, but deem the pre-existing variants inappropriate and would prefer a specialized tool such as the demonstrator.

The features currently available are useful for debugging. Especially the professional developers in the first study, who were directly asked about the separate features, stated that each individual feature had its place and was useful. Two of the three members of the second study's experimental group agreed that the tools was intuitive and suitable for debugging, indirectly confirming that the tool was useful in its current state.

The User Interface is decent, but can be improved. All participants that used the tool stated that the visual semantics and the highlighting made sense to them. The first study's participants confirmed that the new components integrated well with KIELER and their workflow. However, the second study suggests that a better way to manipulate the program's runtime memory is needed.

The quantitative section of the second study showed that the users from the experimental group all appeared to struggle with the tool and its controls, therefore not being able to find many of the bugs and nobody finishing within the fifteen-minute time limit. All three described their struggle with the tool and its usage in the free text section of the survey. Since the professional developers from the first study did not report such problems apart from one of them wishing for an option to lock the diagram view, this may be due to the short time and the fact that the participants were unaccustomed to the tool. To clarify this, a follow-up study could build upon a similar experimental group, but use a control group of people who are unfamiliar with the KIELER simulation. This way, it could be examined whether the new tool is particularly difficult to get used to or whether the control group in this study was simply faster due to their previous experience.

Another aspect of the quantitative analysis is that from the experimental group, only one bug was reported incorrectly by one person and many were reported as not found. Contrarily, in the control group, all bugs were reported as found while half of these reports were actually incorrect. While this may simply be due to the fact that the experimental group's members did not have enough time to even look for all bugs, it may also suggest that using KIELER's simulation and the standard debugger leads users to glance over the code and believing that they found the issue while they have in fact not. This hypothesis is supported by the fact that both members of the control group stated that they found some bugs just by looking at the code, however, another study would need to confirm this with a broader number of participants and a more even start for both groups as described above.

Chapter 7

Conclusion

In this chapter, the problems, proposed concepts and study findings are summarized before giving an outlook at future work for improving the demonstrator implementation.

7.1 Summary

Chapter 1 presented the concepts of debugging in general and model-based languages. It was established that in a model-based scenario, regular debuggers specialize in either the model-based language, disregarding host code surrounding the model in a real application, or the host language, allowing for debugging of both generated and surrounding code. The latter option comes at the cost of not being able to visually examine the state of the source model since the debugger only displays the memory state of the generated host language program. For this purpose, a model-based debugger can be used on the generated code that can help the user relate the generated code's state with the source model it was generated from. Such a debugger should allow the user to set breakpoints on model level, which are then automatically mapped to host language breakpoints in the appropriate code locations. When such a breakpoint is hit or the execution suspends for another reason, the current state of the source model should be extracted from the generated code's runtime memory and displayed to the user on model level.

Chapter 2 presented the family of synchronous languages and in particular SCCharts, which is used as an example language both for explaining language-specific aspects of model-based debugging and for the demonstrator implementation. A previous approach to debugging SCCharts and various compilation approaches for model-based languages were presented. The chapter also outlined concepts for debugging of optimized code and WCET analysis for synchronous languages, both of which rely on compiler tracing, a key functionality exploited for model-based debugging. As an alternative way of visualizing a program's state, a method for algorithm animation was presented. The topic of formal compiler verification as a means to avoid debugging generated code altogether discussed, as well as the option of semi-automatically generating debuggers to reduce implementation effort. Finally, a work on debugging of heterogeneous projects consisting of multiple modelling languages was shown.

In Chapter 3, the Eclipse IDE, the KIELER platform and its model-based compiler were introduced. The concept of model-based debugging was introduced in detail in Chapter 4. Preliminary design decisions such as desired semantics for model-level breakpoints, ways to persist the compiler tracing information in the generated code and options to ensure access to the source model at runtime were discussed first. For SCCharts, three types of breakpoints with different semantics were introduced to ensure that a large enough variety is available for all application scenarios while keeping the complexity manageable. The tracing information is persisted using marker comments to keep the runtime memory clean of them, facilitating formal verification and improving performance. To access the source model, a variable must be used to make it available even when the matching editor is currently unavailable.

Following these decisions, a UI was designed to facilitate setting of breakpoints, displaying the runtime state of the model and controlling the debug session's flow. The source model is displayed to the user and highlighted with runtime information using a visual semantics that is familiar to experienced users from other contexts. The keybindings for setting breakpoints and the control of the debug flow have been designed to be as similar to pre-existing debuggers as possible to make them intuitive. Using SCCharts as an example, it was shown how marker comments with tracing information can be created at compile time and how these markers can be used to place breakpoints automatically and to extract runtime information on active and executing model elements from the generated code's runtime memory. Breakpoint locations are determined using regular expressions scanning the code for matching marker comments, as well as using the AST of the editor. Active states can be determined from the runtime memory of the program directly while executing elements are located using the editor, regular expressions and marker comments. Finally, the aspects specific to the state-based code generation approach were revisited and possible adaptations for both priority-based and netlist-based compilation approaches were presented. The concepts presented here can be applied to the other code generation approaches, too. However, developing the adaptations for priority-based and netlist-based code confirmed that state-based code is clearly the most suitable of the three for model-based debugging and it was thus a good choice to implement the demonstrator for that approach.

Chapter 5 gave implementation details of the demonstrator created for SCCharts. Plugin and code structure, marker comment format and background listeners used to synchronize diagram view and active editor were discussed. The code for the model-based debugger has been integrated into pre-existing KIELER plugins and some components can be reused for other compilation approaches or host languages. Marker comments were designed to both be human-readable and uniquely identify a model element, therefore they include a hash over the model element's fully qualified name and the element name itself.

A set of listeners is used to keep track of events that require reactions from the debugger. When the debug session is suspended, a listener starts the process of extracting and displaying the current memory state. To keep the active editor and the diagram view synchronized, a `PartListener` is used. Since changes to the workspace's resource tree may require reloading of source models or adjustment of breakpoint positions, these events are tracked, too. Highlightings and breakpoints are managed per model to ensure that if multiple models run in the same debug session, their respective data is cleanly separated. Some of the described implementation details have been added later after receiving feedback, or the need for them was discovered during testing. If the code structure and required use cases had been planned out in more detail before implementing, the tool could have been better from the start, allowing for earlier evaluation and thus potentially more study participants.

In Chapter 6, two separate studies were presented that have been conducted to evaluate useability and intuitiveness of the demonstrator. The first study was conducted with two professional SCCharts developers who used the demonstrator as part of their everyday workflow before answering qualitative questions on useability, features, performance and stability. In the second study, participants were given an intentionally flawed SCCharts model, its hostcode environment and bug reports describing the flaws. One group was given the model-based demonstrator while a control group used a standard Java debugger and the pre-existing SCCharts simulation. Both studies suggested that the model-based debugger is superior to the pre-existing options and that participants prefer using it in the future, even though some technical improvements and an improved way of interacting with the model's runtime memory are desirable. They also confirmed that the diagram highlighting is intuitive and the different types of breakpoints are useful to them. Due to the low number of participants in both studies, the results should be confirmed or refuted using a followup study with more participants.

Figure 7.1. KIELER simulation variables view. A similar feature could be implemented for the model-based debugger in the future.

All in all, design, implementation and evaluation have shown that model-based debugging is a viable concept to find errors in model-based code. The most challenging task when implementing the demonstrator was to choose a code structure that would allow for a maximum of reuseability while still being intuitive and following best practices of software engineering. While the usage of KIELER-internal components has been comparably easy, largely due to personal support of the KIELER development team, getting my code to work with Eclipse-internal components has been challenging at times.

The biggest challenge, however, was designing the second study. Here, I had to create a model that would be complex enough to keep participants from spotting errors at first glance while being easy enough to debug within fifteen minutes, using a tool the participants never used before. An introductory video needed to be made that would make the structure and assignment clear without giving away any of the bugs; the questionnaire had to be brief to keep the total time manageable for participants while covering all important aspects. Had there been more time for this process, the study could have been designed even more carefully, which would perhaps have lead to more balanced results, especially in the quantitative section.

7.2 Future Work

Even though the study participants confirmed that the demonstrator is suited for the task of finding bugs in mixed model-based and host-language code, they gave feedback on what could be improved. This section presents these points in detail and how they can be adressed in the future. Alongside this feedback, other ideas for extensions and improvements are presented.

7.2.1 Better Access to Runtime Variables

The main feedback from the second study was that setting input values for the SCChart using the standard Eclipse variables view was tedious. In fact, some participants stated that this was the main reason they ran out of time to find all bugs. While this may improve over time as the users get more accustomed to the memory structure and the process of setting variables with the provided means, the issue of variables being set after they are read due to the scheduling order determined at compile time remains. To tackle this issue and to become generally more user-friendly, the demonstrator should provide a better way of reading and writing runtime variables.

For the SCCharts simulation already present in KIELER, a custom view to show and edit the variables of the model under simulation is available. This view can be seen in Figure 7.1. It allows to

see not only the current, but also past values for each variable. Boolean variables can be toggled by simply clicking their field in the `User value` column, others can have their value entered there.

Using a similar custom view for debugging would make accessing and setting runtime variables much easier than the default Eclipse one. Displaying live values at any given time where the debug session is suspended can be achieved by reading the corresponding values from runtime memory; keeping track of past values may be hard to impossible. To ensure that setting a variable value by hand has the expected effect, a breakpoint can be placed on the first line of the SCChart that is executed. When this breakpoint is hit, the memory state of the program can be set to the desired value by the program before resuming the execution. While this suspension does not need to be visible to the user, it ensures that each variable edit is executed at the beginning of the next tick.

The issue of variables being edited during a tick, but not having an effect until the next one cannot be solved this way. To circumvent this problem, an option to suspend the execution manually at the start of the next tick could be introduced. This way, a user can then set variable values and immediately see their effects, much like they can in the SCCharts simulation.

7.2.2 Improve Linking of Editor and Diagram View

In the studies, some participants reported that the link between editor and diagram view was either unstable, sometimes not updating properly, or behaved unexpectedly. Especially when changing from generated code to source SCChart, they would prefer if the diagram view retained the SCChart it currently displays. To this end, a `Lock diagram` option could be introduced that simply keeps the current diagram in the view, regardless of editor changes. This way, users can decide themselves whether they want the diagram view to be linked to the editor or not. More testing should be done to catch corner cases in which the linking does not currently work as expected.

7.2.3 Support for Other Compilation Approaches and Host Languages

As presented in Sections 4.8 and 4.9, model-based debugging is possible for other compilation approaches, too. Extending the tool to support these use cases would make it more flexible and allow the debugging of less human-readable, but more performant code, too. For maximum compatibility, the demonstrator should also support other host languages, particularly C. C is commonly used for embedded applications, making it the prime host language for SCCharts designed for such scenarios. With a model-based debugger supporting C, such applications could be debugged more effectively. Remote debugging features of C debuggers such as gdb[1] could even support model-based debugging of code running on the target system.

7.2.4 Additional Information in the Diagram

Currently, the diagram view only displays the diagram, along with active and executing states as well as breakpoint markers. However, other information may be useful to the user, too. For example, if a model is instantiated multiple times, the runtime state for each one is visualized in the same view. If they tick sequentially, this means that for each tick, the user will see multiple visualizations with different states, leading to possible confusion. This issue can be mitigated by adding an *instance identifier* to the diagram, e. g., as a suffix to the diagram name. With this identifier, the user can more easily distinguish between values belonging to different instances of the same model.

[1]http://gcc.gnu.org/onlinedocs/gcc-4.6.4/gnat_ugn_unw/Remote-Debugging-using-gdbserver.html

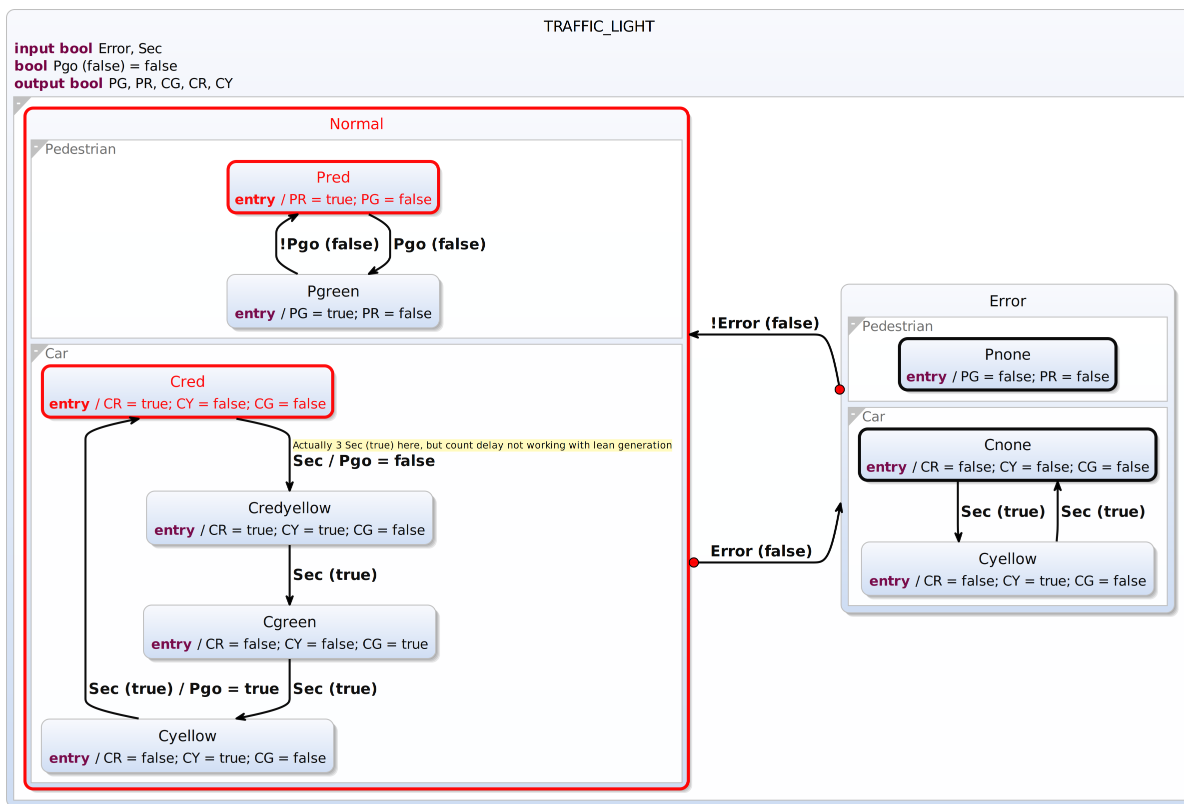


Figure 7.2. Simulated SCChart with live values in the diagram. A similar feature could be useful for the model-based debugger.

To further improve accessibility of the memory state, *live values* could be displayed in the diagram. This option already exists for the SCCharts simulation and can be seen in Figure 7.2. As can be seen there, variables will have their current values annotated in parentheses wherever the variable name occurs in the diagram, e. g., in declarations or transition triggers. With this feature available in the `DebugDiagramView` as well, examining the memory state would become even easier.

Appendix A

Debugging Cheatsheet

The following document has been given to study participants working with the demonstrator tool to remind them of the available features and their semantics, if necessary.

Visual Debugging Cheatsheet

This cheatsheet gives a brief overview of functionality available in the new visual debugging tool.

Debug Diagram View

The debug diagram view is the heart of the new tool. It will display the diagram along with breakpoints and highlightings whenever a suitable file is opened in the Java editor.
It can be **opened** by entering *Debug Diagram* into Eclipse's Quick Access field (the magnifying glass icon in the top right corner) and selecting *Debug Diagram (KIELER)* from the drop-down menu.
The view will automatically display the appropriate model along with any breakpoints set previously.

Setting and Removing Breakpoints

Breakpoints on states:

▻ Set and removed by double-clicking a state

▻ Indicated by a **blue** circle in the top-left corner of the state

▻ Triggered **once** when the state is entered

▻ Only triggered again after the state is left, then re-entered

Transition Taken Breakpoints:

▻ Set and removed by double-clicking a transition

▻ Indicated by a **blue** circle near the middle of the transition

▻ Triggered **once** when the transition is taken

Transition Check Breakpoints:

▻ Set and removed by holding **shift** and double-clicking a transition

▻ Indicated by a **yellow** circle near the middle of the transition

▻ Triggered **every time** the transition's guard is evaluated

NOTE: Due to a known bug, shift + double-click will currently be interpreted as a simple double-click at the same time, thus placing a Transition Taken breakpoint as well. As a temporary workaround, it is necessary to remove that Transition Taken breakpoint by hand by double-clicking the transition again (without holding shift).

Diagram Highlighting

▻ States currently active on SCCharts level have a **red foreground** (i.e. red text and border)

▻ States currently being executed on Java level (a subset of active states) have a **green background** (approximately the same color as the current instruction pointer in the Java editor)

Stepping

▻ The tool supports regular stepping mechanisms of the Java debugger

▻ Use the *step into*, *step over* and *step return* buttons in the top toolbar or the corresponding hotkeys (F5 through F7 by default).

▻ The diagram will highlight the appropriate model elements as for breakpoints

▻ SCCharts-specific step sizes (e.g. skip tick) are not available.

Appendix B

Questionnaire for Professional Developers

This questionnaire has been handed out to professional SCCharts developers after they used the demonstrator implementation for a week to evaluate it. The space for remarks after each section has been removed to better accommodate the questionnaire in this thesis.

Evaluation of Model-Based Debugging

Dear participant,
thank you for taking this brief survey on the model-based debugging approach you have been testing recently. By filling this questionnaire, you greatly help me evaluate my progress and improve the implementation in the future.
Please mark one clear answer for every question provided (unless stated otherwise). If you change your mind after marking a box, please fill it out completely and mark your new answer as usual.
This survey is anonymous and only serves the evaluation of the debugging tool. No personal data is gathered nor processed.

About how many hours have you worked with the tool before filling this survey? (Please round to full hours) ____________________

About how many states did the biggest model you worked with include *before* starting the compilation process? ____________________

What SCCharts features did your models use? (Please check all of them)

☐ Parallel Regions	☐ During Actions	☐ Entry Actions	☐ Exit Actions
☐ History Transitions	☐ Strong Abort Transitions	☐ Weak Abort Transitions	
☐ Count Delay Transitions	☐ Referenced SCCharts	☐ Dataflow Regions	

Useability and Integration	strongly agree	agree	disagree	strongly disagree
The newly introduced IDE components stand out from what I am used to.	☐	☐	☐	☐
I quickly got used to the new functionality.	☐	☐	☐	☐
The highlighting used in the diagram makes sense to me.	☐	☐	☐	☐
I still have trouble remembering what features are available in the new tool.	☐	☐	☐	☐
I want to keep working with the tool in the future.	☐	☐	☐	☐
I am missing important features in the tool.	☐	☐	☐	☐
I prefer using the regular Java debugger over this tool.	☐	☐	☐	☐
I had to change my habits to use the new tool.	☐	☐	☐	☐

Stability and Performance	strongly agree	agree	disagree	strongly disagree
The tool feels slow to use.	☐	☐	☐	☐
I saw error messages that disrupted my workflow.	☐	☐	☐	☐
The tool behaves as I expect it to.	☐	☐	☐	☐
I like the speed at which the tool reacts to inputs and events.	☐	☐	☐	☐
The tool needs more work to become stable before it is ready to use.	☐	☐	☐	☐
The tool makes me wait.	☐	☐	☐	☐
Features	strongly agree	agree	disagree	strongly disagree
I like the features currently available.	☐	☐	☐	☐
I set lots of breakpoints on states.	☐	☐	☐	☐
I don't see why anyone would set (yellow) Transition Check breakpoints.	☐	☐	☐	☐
To work properly, I need more types of breakpoints.	☐	☐	☐	☐
It's not a problem that I cannot set breakpoints on during actions.	☐	☐	☐	☐
All the breakpoints behave as I expected.	☐	☐	☐	☐
(Blue) Transition Taken breakpoints help me understand my model.	☐	☐	☐	☐
I find it useful to have two types of transition breakpoints.	☐	☐	☐	☐
Introducing more types of breakpoints would make the tool too complex.	☐	☐	☐	☐
Sometimes, I wanted to place breakpoints on model elements I couldn't place them on.	☐	☐	☐	☐
I rarely used the stepping functionality of the debugger.	☐	☐	☐	☐
There should be a separate stepping mode to skip the current tick.	☐	☐	☐	☐
The stepping support is in a good place right now.	☐	☐	☐	☐

General remarks

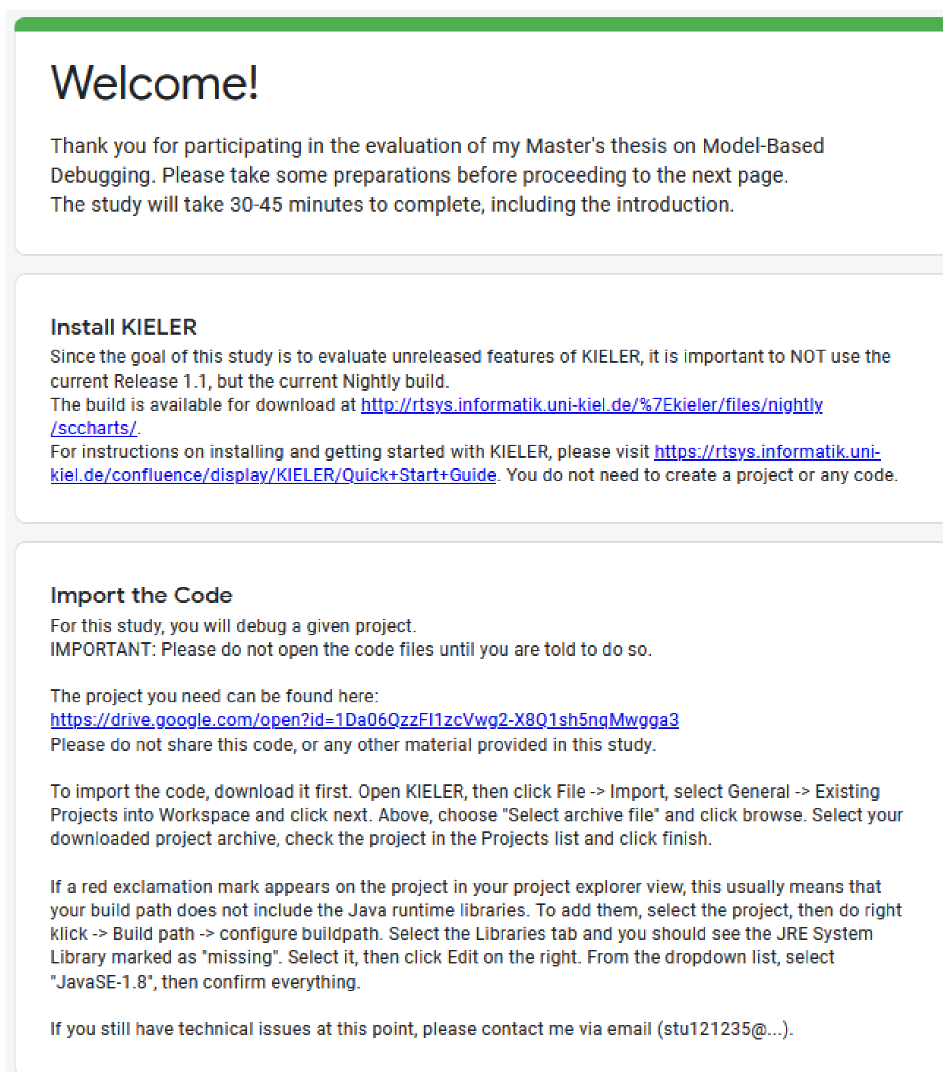
Below, you have extra space for remarks not fitting into any of the above categories. What did you like, what bothered you? What can be improved and what helped you the most? What errors did you encounter while using the tool?

Thank you again for your participation!

Appendix C

Online Survey for University Members

This chapter contains screenshots of the different questionnaire sections used in the online survey conducted with university members. The results and their evaluation can be found in Section 6.2.



Welcome!

Thank you for participating in the evaluation of my Master's thesis on Model-Based Debugging. Please take some preparations before proceeding to the next page.
The study will take 30-45 minutes to complete, including the introduction.

Install KIELER

Since the goal of this study is to evaluate unreleased features of KIELER, it is important to NOT use the current Release 1.1, but the current Nightly build.
The build is available for download at http://rtsys.informatik.uni-kiel.de/%7Ekieler/files/nightly/sccharts/.
For instructions on installing and getting started with KIELER, please visit https://rtsys.informatik.uni-kiel.de/confluence/display/KIELER/Quick+Start+Guide. You do not need to create a project or any code.

Import the Code

For this study, you will debug a given project.
IMPORTANT: Please do not open the code files until you are told to do so.

The project you need can be found here:
https://drive.google.com/open?id=1Da06QzzFl1zcVwg2-X8Q1sh5nqMwgga3
Please do not share this code, or any other material provided in this study.

To import the code, download it first. Open KIELER, then click File -> Import, select General -> Existing Projects into Workspace and click next. Above, choose "Select archive file" and click browse. Select your downloaded project archive, check the project in the Projects list and click finish.

If a red exclamation mark appears on the project in your project explorer view, this usually means that your build path does not include the Java runtime libraries. To add them, select the project, then do right klick -> Build path -> configure buildpath. Select the Libraries tab and you should see the JRE System Library marked as "missing". Select it, then click Edit on the right. From the dropdown list, select "JavaSE-1.8", then confirm everything.

If you still have technical issues at this point, please contact me via email (stu121235@...).

Figure C.1. Welcome page with instructions for KIELER installation and code import.

Group assignment

This section only serves to assign you a random group. Since Google Forms does not directly support random groups, the question below will redirect you to the appropriate section by using a randomized answer ordering.

Please select the FIRST option below to randomly assign you a group. *

- %%
- $$

Figure C.2. Group assignment page of the study. Symbols were used to not give away which group has what purpose.

Preparation

Please ensure that you completed the following steps before proceeding to the next section.
- You installed the current KIELER nightly build
- You downloaded the provided code and imported it into KIELER
- You have access to a timer that can be used to measure your time as well as alert you after 15 minutes. You can either use a smartphone or an online timer such as https://webuhr.de/timer-auf-15-minuten/.

Figure C.3. Confirmation page to ensure that all preparations have been taken

Introduction

Please watch the video below before proceeding to the next section.

Study introduction (German)

Links

Debugging Cheatsheet: https://drive.google.com/open?id=1j_TpEH1vQtN3vK3UCNZIX2JwJWe17cg_

PLEASE START YOUR TIMER BEFORE LOOKING AT THE BUG REPORTS.
List of Bug Reports: https://drive.google.com/open?id=1c-WTAZSCWjzMiulDbvS7DCrR6FevFPth

After reading the bug reports, you may now open the code and start debugging.

Figure C.4. Group-specific page with instructional video and links.

```
#resource "src/AlarmSystemEnvironment.java"

scchart AlarmSystem {

  /** Present once a second */
  input signal second
  /** Indicates that a fire that has been active is now out */
  input signal fireOut

  /** Array of smoke detectors, one per room */
  input bool fireSensors[5]
  /** Array of motion detectors, one per room */
  input bool motionDetectors[5]

  /** Indicates whether it is day or night */
  input bool day

  /** Indicates whether a fire is currently active */
  bool fire = false
```

```
20  /** Indicates whether there currently is an intruder */
21  bool intruder = false
22
23  /** Count how long the lights in each room have been on */
24  int lightTimers[5]
25  /** Indicates whether the light timer in each room should be ticking */
26  bool enableLightTimer[5]
27
28  /** Controls the lights in each room */
29  output bool lightsOn[5]
30  /** Controls sprinklers in each room */
31  output bool engageSprinklers[5]
32  /** Controls an alarm siren used for both intruders and fire alarm */
33  output bool alarmSound
34
35  /* References to external hostcode functions defined by the environment */
36  @hide extern @Java "AlarmSystemEnvironment.callFireFighters" callFireFighters
37  @hide extern @Java "AlarmSystemEnvironment.callPolice" callPolice
38  @hide extern @Java "AlarmSystemEnvironment.tickLightTimers" tickLightTimers
39  @hide extern @Java "AlarmSystemEnvironment.checkMotionDetectors" checkMotionDetectors
40  @hide extern @Java "AlarmSystemEnvironment.disableLights" disableLights
41  @hide extern @Java "AlarmSystemEnvironment.checkFireSensors" checkFireSensors
42  @hide extern @Java "AlarmSystemEnvironment.fireSprinklers" fireSprinklers
43  @hide extern @Java "AlarmSystemEnvironment.disableSprinklers" disableSprinklers
44  @hide extern @Java "AlarmSystemEnvironment.checkForIntruders" checkForIntruders
45
46  /* Always have active light timers running */
47  during do tickLightTimers(enableLightTimer, lightTimers, lightsOn)
48
49  /**
50   * Region to control the overall status of the system.
51   * Day and night mode control whether motion detectors should be used
52   * for switching lights or for detecting intruders.
53   */
54  region SystemStatus {
55
56    initial state Day
57    if fire abort to Fire
58    if !day go to Night
59
60    state Night
61    if fire abort to Fire
62    if intruder abort to Intruder
63    if day go to Day
64
65    state Fire {
66      entry do callFireFighters()
67    }
68    if !fire go to Day
69
70    state Intruder {
71      entry do callPolice()
72    }
73    if !intruder go to Night
74
75  }
76
77  /**
78   * Region to control lights during the day.
```

```
 * Whenever the motion detector in a room is triggered, said room's lights
 * should turn on for a while, then back off after the sensor has not been triggered for a while.
 * At night, lights are disabled.
 */
region Lights {
  initial state lightsEnabled {
    during do checkMotionDetectors(motionDetectors, lightsOn, enableLightTimer, lightTimers)
  }
  if !day go to lightsDisabled

  state lightsDisabled {
    entry do disableLights(enableLightTimer, lightsOn)
  }
  if day go to lightsEnabled
}

/**
 * Region to detect intruders during the night.
 * When a motion detector is triggered at night, the alarm goes off.
 * When the intruder has been caught, the system must be switched to day mode,
 * then back to night mode to disable the intruder alarm.
 */
region IntruderDetection {
  initial state alarmDisabled
  if !day go to alarmEnabled

  state alarmEnabled {
    during do intruder |= checkForIntruders(motionDetectors)
  }

  if day do intruder = false go to alarmDisabled
}

/**
 * Region to detect a fire.
 * When a smoke detector is triggered, firefighters are alerted and
 * sprinklers in that area are engaged. Sprinklers remain on until
 * firefighters report fireOut.
 */
region FireDetection {
  initial state noFire {
    during do fire |= checkFireSensors(fireSensors)
  }
  if fire abort to Fire

  state Fire {
    during do fireSprinklers(fireSensors, engageSprinklers)
  }
  if fireOut do disableSprinklers(engageSprinklers) go to noFire
}

/**
 * Region to produce an alarm sound. If either fire or intruder are present,
 * the alarm sound beeps once a second. When neither is present,
 * the alarm is off.
 */
region AlarmSound {
  initial state NoAlarm
  if fire || intruder go to Alarm
```

```
138
139     state Alarm {
140       initial state SoundOff
141       if second do alarmSound = true go to SoundOn
142
143       state SoundOn
144       if second do alarmSound = false go to SoundOff
145     }
146     if !fire || !intruder do alarmSound = false go to NoAlarm
147   }
148
149 }
```

Listing C.1. Full source code of `AlarmSystem` SCChart used in the study.

```
1  /**
2   * Environment for the AlarmSystem SCChart used for
3   * a study on model-based debugging.
4   *
5   * THIS ENVIRONMENT AND THE MODEL CONTAIN INTENTIONAL ERRORS.
6   *
7   * @author stu121235
8   */
9  public class AlarmSystemEnvironment {
10
11   /** Time in seconds that the light should be on after the sensor was triggered */
12   public static final int LIGHT_ON_TIME = 5;
13
14   private static AlarmSystem system = new AlarmSystem();
15
16   public static void main(String[] args) {
17
18     /* Instantiate and initialize SCChart */
19     system.reset();
20
21     /* Save last time second was present */
22     long lastSecond = System.currentTimeMillis();
23
24     while(true) {
25
26       /* Check whether a second has elapsed */
27       long currentTime = System.currentTimeMillis();
28       if (currentTime - lastSecond >= 1000) {
29         lastSecond = currentTime;
30         system.iface.second = true;
31       }
32
33       /* Perform tick */
34       system.tick();
35
36       /* reset signals */
37       system.iface.second = false;
38       system.iface.fireOut = false;
39     }
40   }
41
42   /**
43    * Calls the firefighters.
44    */
```

```
45  public static void callFireFighters() {
46    System.out.println("Calling firefighters...");
47  }
48
49  /**
50   * Calls the police.
51   */
52  public static void callPolice() {
53    System.out.println("Calling police...");
54  }
55
56  /**
57   * Checks whether any motion detector has been triggered.
58   * If so, trigger intruder alert.
59   */
60  public static boolean checkForIntruders(boolean[] sensors) {
61    for (int i = 0; i <= sensors.length - 1; i++) {
62      if (sensors[i]) {
63        return true;
64      }
65    }
66    return false;
67  }
68
69  /**
70   * Checks whether any fire sensor has been triggered.
71   * If so, trigger fire alarm.
72   */
73  public static boolean checkFireSensors(boolean[] sensors) {
74    for (int i = 0; i <= sensors.length - 1; i++) {
75      if (sensors[i]) {
76        return true;
77      }
78    }
79    return false;
80  }
81
82  /**
83   * Engage sprinklers in all areas where smoke is present.
84   */
85  public static void fireSprinklers(boolean[] sensors, boolean[] sprinklers) {
86
87    for (int i = 0; i < sensors.length - 1; i++) {
88      if (sensors[i]) {
89        // Note that sprinklers are never disabled here.
90        // This only happens if the fireOut signal occurs.
91        sprinklers[i] = true;
92      }
93    }
94  }
95
96  /**
97   * Disable all sprinklers.
98   */
99  public static void disableSprinklers(boolean[] sprinklers) {
100   for (int i = 0; i <= sprinklers.length - 1; i++) {
101     sprinklers[i] = false;
102   }
103 }
```

```
104
105   /**
106    * Turn off all lights.
107    * Also stop all light timers since they are no longer needed.
108    */
109   public static void disableLights(boolean[] enabled, boolean[] lights) {
110     for (int i = 0; i < enabled.length; i++) {
111       enabled[i] = false;
112       lights[i] = false;
113     }
114   }
115
116   /**
117    * Check whether any lights should be turned off.
118    * This should occur if they have been on for longer than
119    * @link{AlarmSystemEnvironment#LIGHT_ON_TIME} seconds.
120    */
121   public static void tickLightTimers(boolean[] enabled, int[] timers, boolean[] lightsOn) {
122     for (int i = 0; i < enabled.length; i++) {
123       if (enabled[i] && timers[i] > LIGHT_ON_TIME) {
124         enabled[i] = false;
125         lightsOn[i] = false;
126       }
127     }
128   }
129
130   /**
131    * Check whether any lights should be turned on.
132    * If so, turn on the respective lights and start a timer
133    * to ensure that they are turned off after an appropriate time.
134    */
135   public static void checkMotionDetectors(boolean[] mds, boolean[] lightsOn, boolean[] enableLightTimer, int[]
        lightTimers) {
136     for (int i = 0; i <= mds.length - 1; i++) {
137       if (mds[i]) {
138         lightsOn[i] = true;
139         enableLightTimer[i] = true;
140         lightTimers[i] = 0;
141       }
142     }
143   }
144
145 }
```

Listing C.2. Full code of `AlarmSystemEnvironment` used in the study.

Bibliography

[And95] Charles André. *Synccharts: a visual representation of reactive behaviors*. Tech. rep. Université de Nice-Sophia Antipolis, Oct. 1995.

[App98] Andrew W. Appel. “SSA is functional programming”. In: *SIGPLAN Not.* 33.4 (Apr. 1998), pp. 17–20. ISSN: 0362-1340.

[BBD+17] Timothy Bourke, Lélio Brun, Pierre-Évariste Dagand, Xavier Leroy, Marc Pouzet, and Lionel Rieg. “A formally verified compiler for lustre”. In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2017. Barcelona, Spain: Association for Computing Machinery, 2017, pp. 586–601. ISBN: 9781450349888. DOI: 10.1145/3062341.3062358. URL: https://doi.org/10.1145/3062341.3062358.

[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 Esterel v5 language primer, version v5_91*. ftp://ftp-sop.inria.fr/esterel/pub/papers/primer.pdf. Centre de Mathématiques Appliquées Ecole des Mines and INRIA. 06565 Sophia-Antipolis, 2000.

[BG92] Gérard Berry and Georges Gonthier. “The esterel synchronous programming language: design, semantics, implementation”. In: *Sci. Comput. Program.* 19.2 (Nov. 1992), pp. 87–152. ISSN: 0167-6423. DOI: 10.1016/0167-6423(92)90005-V. URL: https://doi.org/10.1016/0167-6423(92)90005-V.

[Cop94] Max Copperman. “Debugging optimized code without being misled”. In: *ACM Trans. Program. Lang. Syst.* 16.3 (May 1994), pp. 387–427. ISSN: 0164-0925. DOI: 10.1145/177492.177517. URL: https://doi.org/10.1145/177492.177517.

[DF07] Christopher Davey and Jon Friedman. “Software systems engineering with model-based design”. In: *Proceedings of the 4th International Workshop on Software Engineering for Automotive Systems*. SEAS ’07. USA: IEEE Computer Society, 2007, p. 7. ISBN: 0769529682. DOI: 10.1109/SEAS.2007.9. URL: https://doi.org/10.1109/SEAS.2007.9.

[DPL16] Verislav Djukić, Aleksandar Popović, and Zhenli Lu. “Run-time code generators for model-level debugging in domain-specific modeling”. In: *Proceedings of the International Workshop on Domain-Specific Modeling*. DSM 2016. Amsterdam, Netherlands: Association for Computing Machinery, 2016, pp. 1–7. ISBN: 9781450348942. DOI: 10.1145/3023147.3023148. URL: https://doi.org/10.1145/3023147.3023148.

[EV06] Sven Efftinge and Markus Völter. “Oaw xtext: a framework for textual dsls”. In: *Workshop on Modeling Symposium at Eclipse Summit* 32 (Jan. 2006).

[GHJ+94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional Computing Series. Pearson Education, 1994. ISBN: 9780321700698. URL: https://books.google.de/books?id=6oHuKQe3TjQC.

[Gri16] Lena Grimm. “Debugging SCCharts”. https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/lgr-bt.pdf. Bachelor’s thesis. Kiel University, Department of Computer Science, Sept. 2016.

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

[HDM+13] 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*. Technical Report 1311. ISSN 2192-6247. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Dec. 2013.

[HDM+14] Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer, and Owen O'Brien. "SCCharts: Sequentially Constructive Statecharts for safety-critical applications". In: *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. Edinburgh, UK: ACM, June 2014, pp. 372–383.

[HMA+14] 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". In: *ACM Transactions on Embedded Computing Systems, Special Issue on Applications of Concurrency to System Design* 13.4s (July 2014), 144:1–144:26.

[JHR+08] Lei Ju, Bach Khoa Huynh, Abhik Roychoudhury, and Samarjit Chakraborty. "Performance debugging of esterel specifications". In: *Proceedings of the 6th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. CODES+ISSS '08. Atlanta, GA, USA: ACM, 2008, pp. 173–178. ISBN: 978-1-60558-470-6. DOI: 10.1145/1450135.1450175. URL: http://doi.acm.org/10.1145/1450135.1450175.

[KCS09] Naveen Kumar, Bruce Childers, and Mary Lou Soffa. "Transparent debugging of dynamically optimized code". In: *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. CGO '09. USA: IEEE Computer Society, 2009, pp. 275–286. ISBN: 9780769535760. DOI: 10.1109/CGO.2009.28. URL: https://doi.org/10.1109/CGO.2009.28.

[Ler09] Xavier Leroy. "Formal verification of a realistic compiler". In: *Commun. ACM* 52.7 (July 2009), pp. 107–115. ISSN: 0001-0782. DOI: 10.1145/1538788.1538814. URL: https://doi.org/10.1145/1538788.1538814.

[LKV11] Ricky T. Lindeman, Lennart C.L. Kats, and Eelco Visser. "Declaratively defining domain-specific language debuggers". In: *Proceedings of the 10th ACM International Conference on Generative Programming and Component Engineering*. GPCE '11. Portland, Oregon, USA: Association for Computing Machinery, 2011, pp. 127–136. ISBN: 9781450306898. DOI: 10.1145/2047862.2047885. URL: https://doi.org/10.1145/2047862.2047885.

[Mea55] G. H. Mealy. "A method for synthesizing sequential circuits". In: *The Bell System Technical Journal* 34.5 (1955), pp. 1045–1079.

[MS94] Sougata Mukherjea and John T. Stasko. "Toward visual debugging: integrating algorithm animation capabilities within a source-level debugger". In: *ACM Trans. Comput.-Hum. Interact.* 1.3 (Sept. 1994), pp. 215–244. ISSN: 1073-0516. DOI: 10.1145/196699.196702. URL: https://doi.org/10.1145/196699.196702.

[OR14] Alessandro Orso and Gregg Rothermel. "Software testing: a research travelogue (2000–2014)". In: *Proceedings of the on Future of Software Engineering*. FOSE 2014. Hyderabad, India: Association for Computing Machinery, 2014, pp. 117–132. ISBN: 9781450328654. DOI: 10.1145/2593882.2593885. URL: https://doi.org/10.1145/2593882.2593885.

[Pei17] Lars Peiler. “Priority-based compilation of SCCharts”. https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/lpe-mt.pdf. Master thesis. Kiel University, Department of Computer Science, Oct. 2017.

[SDH19] Steven Smyth, Sören Domrös, and Reinhard von Hanxleden. *A case-study on manual verification of state-based source code generated by KIELER SCCharts*. Technical Report 1905. ISSN 2192-6247. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Dec. 2019.

[SHM+18] Alexander Schulz-Rosengarten, Reinhard von Hanxleden, Frédéric Mallet, Robert de Simone, and Julien Deantoni. *Time in SCCharts*. Technical Report 1805. ISSN 2192-6247. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, July 2018.

[SMH18] Steven Smyth, Christian Motika, and Reinhard von Hanxleden. “Synthesizing manually verifiable code for statecharts”. In: *Proc. Reactive and Event-based Languages & Systems (REBLS '18), Workshop at the ACM SIGPLAN conference on Systems, Programming, Languages and Applications: Software for Humanity (SPLASH)*. Boston, MA, USA, Nov. 2018.

[SMS+19] Steven Smyth, Christian Motika, Alexander Schulz-Rosengarten, Sören Domrös, Lena Grimm, Andreas Stange, and Reinhard von Hanxleden. *SCCharts: the mindstorms report*. Technical Report 1904. ISSN 2192-6247. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Dec. 2019.

[SSH13] Christian Schneider, Miro Spönemann, and Reinhard von Hanxleden. “Just model! – Putting automatic synthesis of node-link-diagrams into practice”. In: *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '13)*. San Jose, CA, USA: IEEE, Sept. 2013, pp. 75–82. DOI: 10.1109/VLHCC.2013.6645246.

[SSH18] Steven Smyth, Alexander Schulz-Rosengarten, and Reinhard von Hanxleden. “Towards interactive compilation models”. In: *Proceedings of the 8th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2018)*. Vol. 11244. LNCS. Limassol, Cyprus: Springer, Nov. 2018, pp. 246–260.

[Sta19] Andreas Stange. “Model checking for SCCharts”. https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/aas-mt.pdf. Master's thesis. Kiel University, Department of Computer Science, May 2019.