Mode Diagram Extraction from Blech Code

Daniel Lucas

Master's Thesis
April 2021

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

Advised by
M.Sc. Alexander Schulz-Rosengarten
In Cooperation with:
Robert Bosch GmbH
Dr. Friedrich Gretz &
Franz-Josef Grosch

Selbstständigkeitserklärung

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

Kiel,

Abstract

Software visualization tools can improve the software development process by achieving a graphical overview of source code and enhancing collaboration. A young programming language that is a good candidate for developing a visualization that facilitates the understanding of the stateful nature of a program is Blech. Blech is an imperative synchronous programming language for embedded, reactive and safety-critical systems and is developed by Robert Bosch GmbH.

This thesis proposes a concept to automatically extract mode diagrams from given Blech code that illustrates the stateful behavior of Blech, while meeting pragmatic requirements for developer usage. To illustrate the mode diagrams, Sequentially Constructive Statecharts (SCCharts) are used. They are a visual programming language and have a tool chain available for automatically generating configurable layouts.

The main findings of this thesis include that the visualization is helpful to understand the stateful nature of the source code and that it could enhance the collaboration between developers. It is also found, however, that the visual programming language SCCharts can be only used effectively if prior SCCharts knowledge is present for the developers or some support about the semantic meaning of the graphical elements is given. Lastly, the findings strongly indicate that preference on different labeling options is highly subjective.

Acknowledgements

I want to thank Prof. Dr. Reinhard von Hanxleden for always pushing for better solutions and being available for advice on and off the battlefield of writing a thesis. I want to thank Alexander Schulz-Rosengarten for advising the thesis and always being available for advice and feedback. The advisement was a great factor to the success of this thesis.

Furthermore, I want to thank Friedrich Gretz and Franz-Josef Grosch for their insights on their work, feedback on milestones and access to relevant information regarding the Blech programming language.

Contents

List of Figures

List of Tables

Acronyms

ELK	Eclipse Layout Kernel
EPL	Eclipse Public License
KIELER	Kiel Integrated Environment for Layout Eclipse RichClient
MDE	Model Driven Engineering
SCChart	Sequentially Constructive Statechart
SVi	Software Visualization

Chapter 1

Introduction

Software development is more than the process of writing source code. It begins by recording the requirements for the desired software and continues with planning of the software architecture, its components, and its environment. Only then, actual source code is written. The process then includes the generation of documentation. It can be generated with visualization tools and languages such as UML. Creating such models can be a tedious task that takes a lot of effort and time. The purpose of documentation is to offer an overview of the software (components) and providing a help for understanding the code base in different abstraction levels. Documentation is beneficial to use for experienced developers as well as developers that just started out working on a given project.

Code, however, is not static. Code changes over time. The reasons for changing code are manifold. Code improvements, refactoring, new features, customer requests or updated technology are all reasons to change existing code. Documentation that was manually created when the code was firstly written is outdated after the code changes. Hence, the documentation also has to be updated manually. This is a tedious task that has to be remembered to be done. As soon as this task is not executed once, code and documentation are out of sync and all types of issues arise. Therefore, automatic generation of code documentation could enhance the software development process.

Apart from interface documentations, documentation of software is often of visual nature. Generating visual representations of code is generally called Software Visualization and is a broad field of diverse issues [GME05]. Most of the time, graph structures are used to visualize source code. With the help of directed edges, the control flow of a program can be visualized intuitively. Therefore, automatically extracting models from source code and visualizing them is a set task when trying to automatically visualize software.

Visualized software serves more purpose than documentation. Visualized software can enhance collaboration in projects with multiple members or can make it easier to find possible improvements in the code structure [GME05].

Software can be visualized using many different levels of abstraction. The most abstract way software can be viewed is by its requirements. The software itself is a black-box, where only the in- and outputs are known. This is a very high level of abstraction. The lowest level of abstraction is given by the complete code including all operations, as no information on the implementation is missing. Abstraction is a spectrum and any point in between these two extremes is an abstraction level that can possibly be desired to be visualized. Possible abstractions may discard simple calculations and only show the control flow in the program or consider the software components to be black-boxes and only visualize their interactions. The

sought level of abstraction depends on the intention behind the visualization. For example, if the interaction between software components is of interest, an abstraction will be chosen that hides the inner behavior of the software components and highlights their interactions only.

A relatively young programming language that does not have many tools available for automatically visualizing its source code is Blech[1]. Figure 1.1 shows an example of Blech code that was used for the announcement of this thesis. Blech is an imperative synchronous programming language for embedded, reactive, real-time, safety-critical systems that allows to write reactive subprograms, which are combined sequentially and concurrently [GG18]. Blech can be compiled to C and thus, be integrated in existing projects or simulation frameworks. The language differs from its competitors Esterel [Ber00], Quartz [Sch09], Lustre [CPH+87] or Céu [SIR15] in terms of targeting application-level software development with a flat learning curve in industrial contexts, easy project integration and its notion of causality. Unlike older languages with broad applications and many users such as C or Java, it does not have a broad variety of supporting tools available. Apart from the compiler, an extension for Visual Studio Code with a language server for syntax checking and highlighting is available. To use Blech in large-scale software projects, a tool for automatically generating documentation and visualizations for collaborative work could improve its usefulness.

To determine the value of such illustrations, existing visualization tools could be used. This way, not all parts of such project regarding the graphical elements and their layout have to to be developed. A research project about the graphical model-based design of complex-system is the Kiel Integrated Environment for Layout Eclipse RichClient (KIELER). Its pragmatics part aims to improve comprehensibility of diagrams, improve development and maintenance time and improve the analysis of dynamic behavior. Automatic layout is applied to graphical components of models. A visual language illustrated by KIELER is Sequentially Constructive Statecharts (SCCharts), designed for modeling safety-critical reactive systems. SCCharts are based on a statechart notation [HDM+14]. SCCharts are further described in Section 2.4.1 and an example is given in Figure 1.2.

1.1 Defining the Goal

As explained in the previous paragraphs, the goal of this thesis is to visualize Blech code. The goal is to produce a visualization that may serve as documentation or for collaboration when working with Blech. The result, however, should not just be any visualization, but meet certain requirements that are discussed in this section.

The Blech compiler is currently capable of producing a visualization of the program path. The technical details are discussed in Section 2.4.2. An example of this current visualization that was generated using the example Blech code in Figure 1.1 is given in Figure 1.3. The figure shows a control flow graph. This visualization, however, serves as a tool for debugging the compiler. Its goal is not to pragmatically visualize the Blech code. The source code is hard to match to the graphical elements and the meanings of the different shapes and colors

[1] https://www.blech-lang.org/docs/user-manual/

```
activity StopWatchController
  (startStop: bool, resetLap: bool)
  (display: Display)
    var totalTime: int32
    var lastLap: int32
    repeat
        totalTime = 0 //State init
        lastLap = 0
        writeTicksToDisplay(totalTime)(display)
        await startStop // Transition init -> run
        repeat
            cobegin weak
                await startStop
            with weak
                run Measurement(resetLap)
                (totalTime, lastLap, display)
            end
            // State stop, show total time and wait
            writeTicksToDisplay(totalTime)(display)
            await startStop or resetLap
            // Run again if only startStop was pressed
        until resetLap end // Back to init if
    end                    // resetLap was pressed
end
```

Figure 1.1. Blech code showing an activity from a stopwatch project that was used in the announcement of this project by Robert Bosch GmbH.

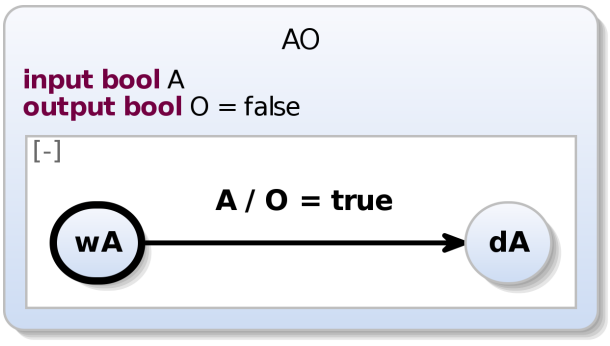


Figure 1.2. An common SCCharts example [Mot17].

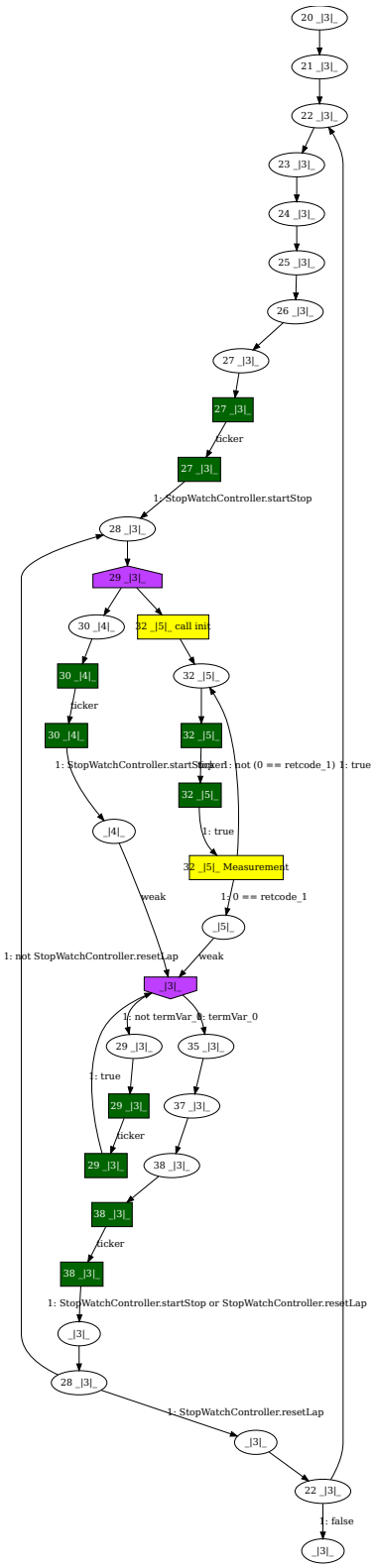


Figure 1.3. The graph description language based control flow graph of the Blech code given by the project announcement in Figure 1.1.

need to be understood to explore the meaning of the visualization. Furthermore, the figure contains more nodes than the represented code has lines. Therefore, it is not a help for quickly understanding the given code base and enabling productive collaboration. A higher level of abstraction is needed. Lastly, it needs to be mentioned that the tool for generating the image offers only limited layout configuration options. The nodes not needed for code comprehension and lack of configuration options result in the large illustration.

Focusing on visualizing the control flow and additionally deleting unnecessary nodes from the visualization would only bring benefits with small effects, as it only shows information that is already explicitly specified in the code. As discussed in the following paragraphs, Blech contains implicit information. Visualizing implicitly contained information while omitting unnecessary code elements that do not enhance the visualization is probably a more satisfying way to give an overview and understanding of the code base and to enable project collaboration. Possible unnecessary elements to be left out are variable declarations or simple expressions that do calculations on these variables. They do not enhance abstract visual representation. If details about these calculations are needed, the code can simply be considered and viewed.

Blech is a programming language for embedded reactive systems and does not explicitly define states or modes of operation, unlike SCCharts, for example. States are implicitly given by (await-)statements instructing the program to wait for defined conditions. Statements are executed in Blech until the program is halted by such an await statement. When the condition of an await statement is met, the program continues to run in another reaction. The condition is checked at the beginning of a tick, so the execution can resume at the earliest in the next tick after the program was halted. The mode of operation has changed, or it can be said that the program has entered a new state. Code that is executed between two await statements belongs to a conceptual state that is bounded by these await statements. A pseudo code example of an automatic door mechanism switching between two modes of operation is given by Algorithm 1. The code contains two blocks separated by two await statements. The two modes of operations or states are a door opening and door closing state, respectively. They are conceptually separated by the two await statements determining the switch between the two distinct contexts.

Algorithm 1: Pseudo code showing a switch between two modes of operation.

```
1 while true do
2     open door (Block 1)
3     await conditionA
4     close door (Block 2)
5     await conditionB
```

The different modes of operation or states are connected via the conditions given by the await statements. Such data structure of connected states is a graph, where the states are the nodes and their connections are the edges or transitions. In general, this is called a

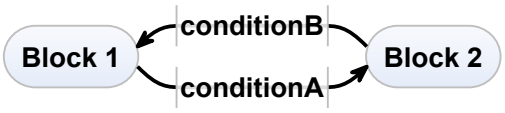


Figure 1.4. Illustration of the implicitly given state machine of Algorithm 1.

finite-state machine [Wan19]. The described graph to the pseudo code given by Algorithm 1 is visualized in Figure 1.4. The two identified states are connected via edges showing the conditions that are given the conditions of the await statements to switch the mode of operation or state.

The visualization of such implicitly contained states is currently not possible for Blech. Since it visualizes information not explicitly given in the code itself, it is probably a helpful visual representation of the code. However, the states depend on surrounding control flow elements. Imagine a block of code with the described stateful behavior that is only executed if a certain condition is met, i. e., an if-instruction evaluated to true. When visualizing the stateful behavior, there should be a visual indication that the behavior is only relevant if the specified condition is met. In general, the visualization should be mindful of the control flow surroundings of the stateful behavior.

All things considered, the desired outcome should target to visualize an abstraction level that neglects unnecessary declaration and calculations, but highlights the stateful nature of the given Blech code while maintaining the needed information of the control flow to correctly show the implicitly contained states. Since documentation, code overview and enhancing collaboration are side targets of the visualization, it should cater to some pragmatic needs, i. e., the drawing should be free of unnecessary nodes and edges that do not convey important information. A smaller sized visualization with only as many nodes and edges as needed allows for a faster assessment of the drawing and thus, enhances collaboration and offers a better overview. Figure 1.3 showed an illustration with multiple nodes and edges not conveying important information.

It might be beneficial to allow a user to define labels for the implicitly given states. As can be seen in Figure 1.1 in line 7 and 18, a name for a state is specified in the comments. The mental map and planed mode of operations by the developer can be preserved better, if such labels are possible. Without such labels, the only indication about the context of the program would be given by conditional transitions through control flow elements or await statements. Since conditions can be of arbitrary complexity, their presence could even be a distracting or confusing factor when determining the context of the states in the visualization. When proposing a concept for the abstract stateful Blech code visualization, the possibility to specify labels to offer the developer a degree of customization of the visualization should be present.

Once a graph with the given requirements is extracted from the Blech code, it needs to be visualized to serve its purpose as documentation or collaboration enhancing tool. As introduced earlier, SCCharts are a visual programming language for modeling safety-critical reactive systems. Since they are based on a statechart notation, have a tool chain available

for visualizing graphs with automatic layout options, and share the same domain as Blech, they are a natural choice to explore possible visualizations of implicitly contained state charts in Blech code. The shared domain of embedded, reactive and possibly safety-critical systems allows to reuse the many graphical SCCharts elements with equivalent semantics. Semantic behavior can be easily highlighted this way. SCCharts further have a set of transitions available to make the described distinction between state connection via await statements and control flow transitions as discussed earlier. Through the available visualization tool chain, customizable drawings are generated and the work of this thesis can primarily focus on the extraction of the needed information from the source code. Compliant to the popular design pattern model-view-control [LR01], the fundamental graph data structure that is to be extracted is independent from the visual representation. SCCharts serve as the view layer and can be replaced in future work, if they are found to be a suboptimal fit.

A key part of this thesis is the implementation of the proposed concept. The extraction of the needed information from the Blech code can take place in the Blech compiler and be activated via flags. The compiler already precompiles the code into an abstract syntax tree. The tree can then be parsed to generate the defined desired graph structure. Unwanted statements can simply be ignored when parsing the tree. Another module in the compiler can then translate the extracted graph structure into textual SCCharts, which can then be visualized by existing SCCharts compatible editors such as KIELER. In future work, the visualization of the code could be displayed in a window in the Visual Studio Code programming environment, which is currently the primarily supported IDE for Blech. With such a visualization in the IDE that shows the illustration in real-time while the developer works on the code, an enhanced development process could be achieved. The SCCharts visualization tools would need to be integrated into either a new Visual Studio Code extension or alternatively into the existing Blech tools.

The summarized goal of this thesis is to find a valid, helpful visual representation of the stateful nature of Blech code. The visualization is to be generated automatically by transforming the Blech code into SCCharts.

1.2 Outline

Chapter 2 first presents a basic overview about synchronous and visual programming languages as well as their general advantages and goals are explained in Section 2.1. Since the extraction of the introduced stateful graph and its translation are integrated into the Blech compiler, the topic of compiler construction and its phases is discussed briefly in Section 2.2. Next, Section 2.3 introduces work related to this thesis. This part covers the model extraction from source code and general software visualization including corresponding development enhancement tools. It is found that most existing tools offer limited help since they are often tailored to specific technologies and languages. Lastly, the chapter introduces the used technologies SCCharts, KIELER, Blech and its available tools, and F#.

Chapter 3 then proposes the conceptual translation from Blech code with the help of graph

structures into SCCharts in detail. The Blech statements and constructs of interest are covered individually. Next, Section 3.2 proposes two labeling concepts to be able to further customize the visualization. After the individual translation steps and labeling, simplifications have to be made, since each statement was translated on its own. Simplification steps of flattening hierarchy and collapsing transient states have to be made to create a transient-state-less and compact drawing. Section 3.3 discusses the simplification steps.

Chapter 4 discusses the implementation details. The data types used for the abstract graph and Blech types offering the details on the abstract syntax tree are discussed. The detailed phases of the implementation are presented. Some specified details that were problematic during implementation are also discussed, such as the presence of SCCharts keywords in variable, activity or function names in Blech. Further, the configuration options introduced by the translation module of the compiler are listed.

Next, the proposed concept and its implementation are evaluated. Chapter 5 discusses the assessment of constructed Blech program examples that cover scenarios of interest as well as real Blech programs taken from the official Blech website. The same Blech programs are given to a group of experts on Blech and statecharts in general. Their assessment serves as the first evaluation for the results of this thesis.

Lastly, Chapter 6 lists the findings and results of the implementation and evaluation. It further discusses debatable points found in earlier chapters. Finally, the chapter concludes the work and introduces some work to be done on this topic in the future.

Chapter 2

Preliminaries

This section introduces basic concepts important to understand the details and context of this thesis. General descriptions on the context in terms of synchronous and visual programming languages, the domain of embedded systems and compiler construction are given. Next, related work of this thesis is presented, categorizing this thesis along the current research findings. Lastly, used technologies are presented.

2.1 Synchronous and Visual Programming Languages

Programming languages are formal languages that are used to specify desired behavior of the computer in form of a program. The languages consist of instructions that can be assembled to transform a given input into a given output. Programming languages differ in terms of their syntax, semantics, domain, type systems, programming paradigms and many other properties. Along these properties, programming languages can be sorted into families. The main languages of concern in this thesis, Blech and SCCharts, can also be classified into programming language families. Two of these families are visual and synchronous programming languages [FVC+13].

Synchronous languages are used for programming reactive systems. Reactive systems are systems interacting with their environment under time restrictions, i. e., the reaction cannot wait. Specific features of synchronous languages include: determinism, concurrency or applicability to distributed architectures. Most of the time reactive systems are also safety-critical, e.g. any system involving the control mechanisms of airplanes. Hence, these systems also have critical reliability requirements. Consequentially, synchronous languages cater to these needs. In synchronous programming the reaction to an event is logically atomic, i. e., takes no time to compute, even though multiple instructions have to be executed. A popular and early formal language specifically designed for specifying and designing reactive systems are Statecharts, which were introduced by Harel [Har87]. Other popular synchronous languages are Lustre [HCR+91] or Esterel [Ber00]. A synchronous language-specific semantic problem is causality. A program is considered to be causal, if it is free of cyclic read-write dependencies [SSH19]. Finding a valid schedule automatically can be problematic [SSH19; Hal98].

Visual programming languages use visual elements in the language to specify the desired behavior. Usually, some sort of graph consisting of nodes and edges is used. These languages are not limited to visual elements as some of them use textual descriptions as well to enhance

the expressiveness and usefulness of the language. Visual elements have been found to be helpful in terms of understanding code and its semantics. Visual programming languages aim to target a broader audience of people who might have trouble programming with textual specification. A common argument for considering visual programming languages are hieroglyphics, one of the earliest forms of written down communications and consisting of visual elements only. A well-known example of a visual programming language is Ptolemy [EJL+03; BD04].

2.2 Compiler Construction

A compiler is a program that translates given code from a source to a target programming language. Usually, a higher-level programming language is translated into a lower-level programming language, e.g. C code into machine code. There are different types of compilers. Translating between to higher-level languages is done by so-called source-to-source compilers [MW97].

Compiler programs are broken down into modules or phases. The phases as described by Maurer and Wilhelm [MW97] are as follows.

1. Lexical Analysis
2. Filtering
3. Syntax Analysis
4. Semantic Analysis
5. Machine-independent Efficiency-enhancing Transformations
6. Adress Assignment
7. Code Generation
8. Machine-dependent Code Optimizations

Some of these phases depend on the source and target language and are not always needed. Generally, the most important phases are the lexical analysis, syntax analysis, semantic analysis and code generation. The phases are shortly described in the following. For details, consider Maurer and Wilhelm, 1997 [MW97].

The lexical analysis takes the program code to be compiled as input. It dissects the program into lexical tokens, symbols or words. A token is a pair of a token name and its value. Token categories that are often found are separators, operators, keywords, identifiers and more. Hence, this phase dissects the program from its character sequence into a sequence of identified symbols. This part of the compiler is also usually called the scanner.

The following filter identifies important reserved elements, such as `int` as a type in most languages, and directives (pragma) for the compiler. It also filters irrelevant symbols. Usually,

it also uniquely encodes symbol classes, such as identifiers, and replaces all occurrences. The result is a filtered sequence of symbols.

The syntactic analysis identifies the structure of the code. It identifies expressions, assignments, declarations and control-flow inducing structures in the code and checks their correctness. This part of the compiler is also usually called the parser. Errors in the syntax of the code are detected in this phase. For example, a type is needed for variable declarations in Java. If there is no such keyword to be found, the parser cannot identify the piece of code as a variable declaration and will most likely throw an error. The sequence of symbols is transformed into a syntax tree in this phase.

In the semantic analysis, static semantic properties are checked. Properties such as type correctness of strongly typed languages, i. e., type checking, consistent type assignment of functions, object binding or definite assignment are checked. The result of this phase is a typed abstract syntax tree.

In the next phase, machine-independent transformations are made to the tree to improve the generated code. Analysis steps take place, such as the generation of a call graph of the program or a dependency analysis. Then optimizations take place, for example dead code elimination or loop transformations. Another simple example of a possible optimization is given as follows: Consider the program variable $a = 2$ and an expression $b = a + 2$. During the compilation, the value a can be inferred and the expression can be evaluated before the dynamic runtime to $b = 2 + 2 = 4$. Sometimes, this phase is disabled for performance reasons and has to be enabled manually.

During the adress assignment, the target machine is considered and information about the memory relevant issues, such as word lengths to be saved, are generated. Target machines can have these memory requirements and have to be considered in this phase.

The code generation then finally produces the target code. If the target language is a low-level programming language, the proximity of often used variables has to be considered for performance reasons. For higher-level programming languages, this is not an issue and the given syntax tree can simply be translated node by node into the corresponding target language elements.

Lastly follows a phase for machine-dependent optimizations steps. This phase tries to identify more unnecessary statements that were produced by previous general translation steps. These steps might also cause instructions that are not optimized for special cases. Such occurrences are detected and fixed in this phase also.

2.3 Related Work

This section introduces some work related to this thesis, either solving similar problems or serving as foundations for this thesis. This section covers not only work about model extraction but also about code visualization and documentation.

2.3.1 Model Extraction

This section considers some works on extraction of models of any form from code.

Model Extraction from C

Model extraction from source code is covered in a master thesis from Andersen [And19]. Abstract models are extracted from C and C++ code in order to serve as documentation of legacy code. Legacy code is often not trustworthily documented, if it is even documented at all. This work proposes a conversion from the extracted model to SCCharts, which are visualized by the KIELER tool chain. Two models are extracted in this thesis. One is for dataflow, highlighting in- and output and changes made to these variables. The second model is a representation of a state-machine for the code with defined states, input events and transitions. These models are the main difference to the work presented in Section 2.3.1.

The presented thesis differs from this thesis in multiple ways. Obviously, the work extracts its models from C and C++ code, which are different from Blech code. Blech code compiles to C code, however, which might offer ways to benefit from the presented thesis. The C code resulting from the compiler is generated code, however, and it most likely does not resemble the given Blech code intuitively. Another difference is the extracted model. The goal of this thesis is not to extract a state-machine with all transitions and states or a dataflow model, but an abstraction of the code aiding to document and understand the code and representing the stateful nature of the code. The generated state machine given by Andersen's work produces a fully functional state machine.

Another Model Extraction from C

Lenga et. al [SLH16] also propose an extraction from C code to SCCharts. They extract an abstract syntax tree from the code and translated the elements of C to corresponding SCCharts structures. They then use existing compilation tools from KIELER to compile the extracted SCCharts back to C and compared the results. Their approach is limited to ANSI C.

Again, this work differs in source code and target model. Yet, similarities to this work's topic are present. This approach differs to the previously presented approach by doing a literal translation of C code to SCCharts without having code comprehension in mind.

Interactive Transformations for Visual Models

Rüegg proposes a concept of a step-wise transformation of Esterel to SyncCharts in his Bachelor thesis [Rüe11]. The focus lies on step-wise transformation because most transformations in the Model Driven Engineering (MDE) domain are exectued in a black-box. A user does not get to comprehend the distinct transformation steps that can be quite complex. The thesis implements a step-wise visualisation. With the step-wise transformation it is also easier to detect errors in the transformation. SyncCharts and Esterel are graphical and textual programming languages, respectively. In the work it is argued that implementing such automatic

transformation enables the the usage of the advantages of textual and graphical languages, respectively. The presented advantages are improved comprehensibility for graphical languages and faster editing for textual languages. The stepwise transformation is implemented in the context of KIELER.

The work of Rüegg differs from this thesis' work. Firstly, this thesis does not primarly aim at a stepwise execution, though having a step-wise execution might be a beneficial feature to implement in the future. Secondly, this thesis does not aim to propose a concept that provides a semantically equivalent model transformation but rather a visual abstraction of the source code. Lastly, the source and target languages differ.

2.3.2 Software Visualization

This section considers tools and approaches for code comprehension and software visualization.

Software Visualization in General

Gračanin et al. [GME05] wrote an article on Software Visualization (SVi). They described what SVi is. It is described as a field that uses visual images to provide insight and understanding and reduction of complexity of existing software (systems). The goal is always better a understanding of given software artifacts. They further describe different research in the field of SVi and recommend further readings for different questions regarding SVi. Many research questions come down to the questions of what can and should be visualized and how, and what benefits does the visualization bring to the developer. Overall, SVi is a broad field with many specific research questions.

Code Visualization with ExplorViz

The Software Engineering research group at Kiel University has developed the tool ExplorViz[1] [FWW+13]. It takes a whole software system as input and creates a so-called 3D city model to display the communications between the different parts of the system. The model itself is interactive so the user can open the boxes representing software components to show its contents. Inside these boxes are either further packages, again displayed as boxes, or classes, which are displayed as blue cylinders.

In addition to the display of the system structure, the tool also displays the communication between different parts of the system. For each time a class calls a function of another class, a yellow line is drawn between the corresponding blue cylinders of the two classes. Hence, components that communicate more regularly are indicated more intensely.

[1] https://www.explorviz.net/

2.4 Used Technologies

As discussed in Chapter 1, this thesis uses SCCharts to abstractly illustrate Blech code. Blech code is translated to textual SCCharts, which are then visualized by KIELER. The Blech compiler module producing the textual SCChart from Blech code is written in F-Sharp (F#). These technologies are portrayed in detail in this section.

2.4.1 SCCharts

Sequentially Constructive Statecharts (SCCharts) are a visual language. They are designed for modeling safety-critical reactive systems. Introduced by von Hanxleden et al. in 2014 [HDM+14], SCCharts use a statechart notation similar to statecharts that were introduced by Harel [Har87]. SCCharts can be viewed as an extension to SyncCharts, which were proposed by André [And95]. Features that can be found in SyncCharts and SCCharts are hierarchy, concurrency, modularity, and others. Furthermore, determinate concurrency based on a synchronous model of computation (MoC) is provided. The semantics of SCCharts are defined on a determinate basis, but the safety-critical focus is also reflected in the definition of the language. Several projects have been completed using SCCharts, such as modeling software for a model railway controller [The06]. Figure 1.2 illustrates a simple example of SCCharts, which illustrates a root state called `AO`, an interface declaration containing boolean input and output variables, and a region. The latter contains two states connected by a transition with a transition trigger and effect. Each region contains exactly one initial state, which is `wA` in this case.

The SCCharts elements are described in Figure 2.1. There are two sets of SCCharts: core and extended. The core set is a small set of simple features for easy compilation, while the extended set is a rich set of advanced features for easier modeling. The core elements are used for the visualization presented in this thesis. From the extended set, connectors, complex final states, conditional termination, strong aborts, weak aborts and pre operators are used. Furthermore, final regions are used, which are not illustrated in this image.

A super state is a hierarchical state with inner behavior. If mutliple regions are present, they are executed concurrently. Each region contains an initial state and might have a final state. Reaching the final state causes the super state to be exitable. Super states can have in- and output variables as well as locally declared values. States are connected via transitions. Transitions can be immediate or delayed, with the first meaning that a transitions can be taken immediately during execution and delayed meaning that the transition can only be taken in the next tick after reaching the source at the earliest. Transitions can be of aborting nature or terminating, i. e., the inner behavior of the source state has to reach its final state, or neither. Lastly, transitions can be strong aborts, strongly aborting the source state. Transitions have triggers, which guard the transitions and have to be met for the transition to fire. They can also have effects. When multiple transitions are available from a source, they are prioritized by priority labels. Lastly, final regions consider all their inner states to be of final nature.

For more details on SCCharts consider the corresponding literature or documentation[2] [HDM+14].

[2] `https://rtsys.informatik.uni-kiel.de/confluence/display/KIELER/SCCharts`

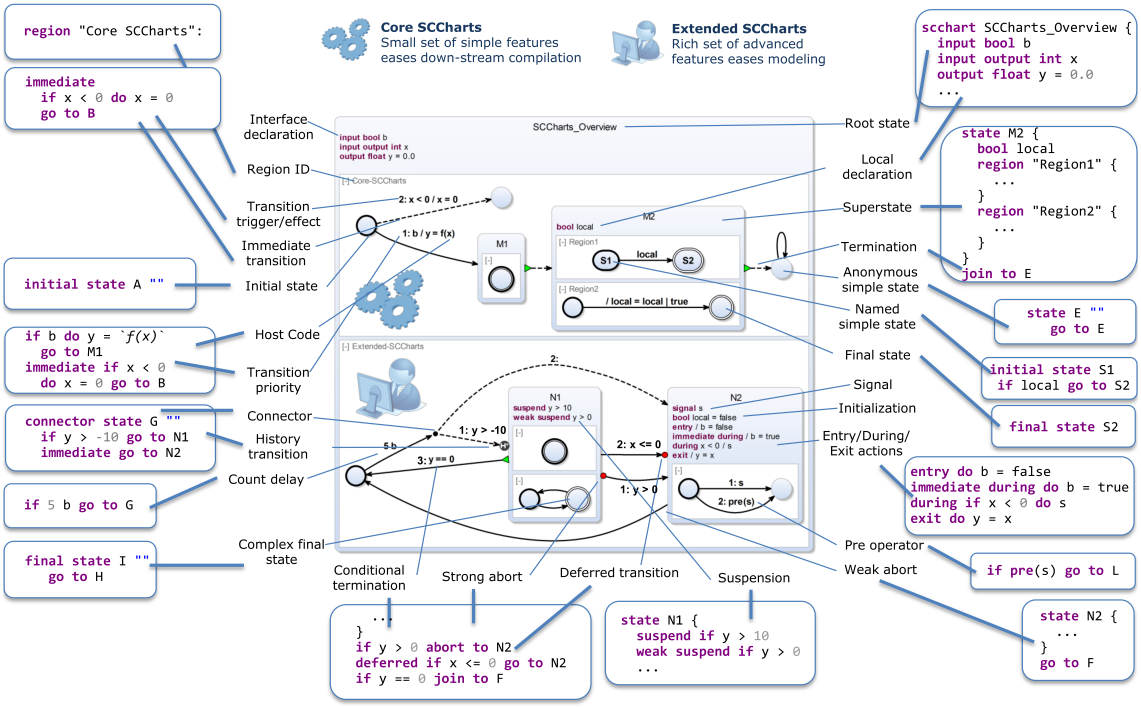


Figure 2.1. Explanation of the visual and their corresponding textual SCCharts elements. There are core and extended SCCharts. Extended ones offer more expressive visualizations. [21].

2.4.2 Blech

Blech is a synchronous imperative programming language for embedded systems and is developed by Robert Bosch GmbH. Blech offers reactive, concurrent programming, guarantees causality and targets application-level software development. The language differs from its competitors Esterel, Quartz, Lustre or Céu in terms of specifically targeting application-level software development, offering a flat learning curve in industrial contexts, and providing a different notion of causality. Blech compiles to C code and can therefore easily be integrated into existing projects. An example of Blech code is given by Figure 2.2 [GG18].

The example code is a part of a program controlling the blinker of a car. It shows important features of the languages, such as including external variables of the surrounding projects, concurrency with its cobegin constructs, aborting conditions, and the halting of a program through the await of a given condition.

Through its interaction with environmental variables, Blech can execute operations as a reaction. The initial reaction starts at the declared entry point activity. A Blech programm may consists of one or more activities, having one or more reacting statements (await statements) each. An await statement holds the execution of a thread until the specified condition is met. Functions are similar to activities and may be used to outsource code into units of concern but are free of stateful behavior. They simply calculate given input into specified output [GG18].

```
singleton activity BlinkerController (warningPushed: bool, blinkerLeverPos: int32)
    @[COutput (binding="actuator_state.blinker_left_on", header="env/blinker_env.h")]
    extern var leftBlinker: bool
    @[COutput (binding="actuator_state.blinker_right_on", header="env/blinker_env.h")]
    extern var rightBlinker: bool
    @[COutput (binding="actuator_state.warning_indicator_on", header="env/blinker_env.h")]
    extern var warningIndicator: bool

    repeat
        when warningPushed abort
            cobegin
                run ConditionalBlinker(blinkerLeverPos, BLINKER_LEVER_POS_UP)(rightBlinker)
            with
                run ConditionalBlinker(blinkerLeverPos, BLINKER_LEVER_POS_DOWN)(leftBlinker)
            end
        end
        when warningPushed abort
            cobegin
                run Blinker(BLINKING_RATIO_WARNING)(warningIndicator)
            with
                repeat
                    leftBlinker = warningIndicator
                    rightBlinker = warningIndicator
                    await true
                end
            end
        end
        warningIndicator = false
        leftBlinker = false
        rightBlinker = false
    end
end
```

Figure 2.2. Example of Blech code showing some language features. This example is taken from a car blinker example from the official Blech website: https://www.blech-lang.org/docs/examples/.

Control flow is specified in Blech via regular programming language elements, such as repeat loops with ending conditions or without, if-else statements or while-loops. Further, Blech allows for declaration of aborting behavior. Nested behavior inside an abort statement will be aborted after the inner execution has halted for the first time and the specified condition is met. Abort constructs in Blech allow to either abort back to the beginning of the inner behavior, i. e., a normal abort surrounded by an endless repeating loop, or skip the inner behavior and continue in the control path [GG18].

Concurrency in Blech can be expressed through cobegin-constructs with an arbitrary number of branches. The declared branches are executed concurrently. Branches may be weak, which means they are aborted if another branch terminates before they do. If all branches are weak, the branch terminating first terminates the other branches [GG18].

When compiling the Blech code, a causality analysis is performed. Concurrent access to shared data is analyzed to determine in what order the data is accessed and if it can be modeled consistently. The analysis will detect programming mistakes for the access of shared data: write-write conflicts and circular read-write dependencies. To correctly model the latter situation, the Blech keyword `prev` may be used, which points to the value of a variable from an earlier tick [GG18].

Activities and functions declare two separate lists: read-only input parameters and read-write output parameters. This way, a programmer can explicitly declare an interface that can easily be checked for causality by the programmer himself, when considering the activities and functions to be black-boxes [GG18].

The currently available tools for the Blech language are written in F#. These tools compile and run on the open-source Microsoft .NET platform. The two tools available are the Blech compiler *blechc*[3] and the Blech Language Services for Visual Studio Code[4]. The compiler compiles given Blech code to C code under given configurations. The language services are a Visual Studio Code extension offering syntax highlighting and a language server for checking the code and giving developer support. For detailed information about compiling and using these tools consider the official Blech website[5].

The Blech compiler can generate graph description language-based control flow graphs per activity and topologically sorted block graphs. The graphs can be generated by enabling the option `-v d` during compilation. To visualize the textual output, the tool *Graphviz* can be used. This was used for the generation of Figure 1.3. Alternatively, web tools can be used to visualize the output[6]. Graphviz visualization tools offer only limited configuration options for layout.

Chapter 4 covers important technical details of the Blech compiler relevant for this thesis.

[3] `https://github.com/boschresearch/blech`

[4] `https://github.com/boschresearch/blech-tools`

[5] `https://www.blech-lang.org/docs/getting-started/`

[6] `https://dreampuf.github.io/GraphvizOnline/`

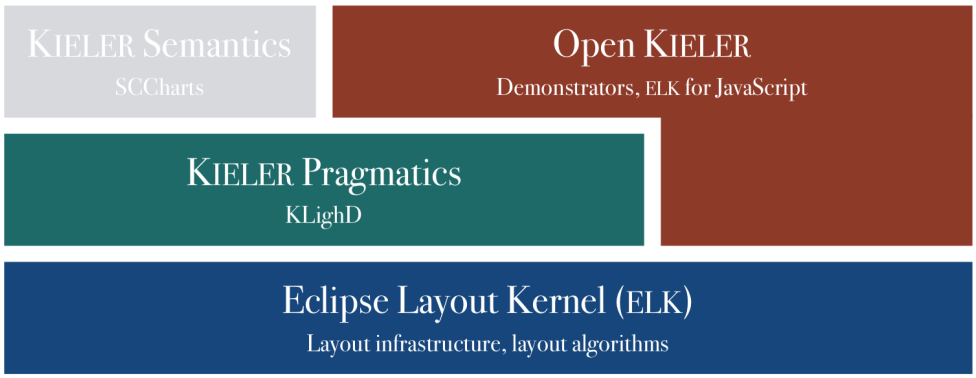


Figure 2.3. Division of the KIELER research project into layout(ELK), pragmatics, semantics, and open KIELER.

2.4.3 KIELER

The Kiel Integrated Environment for Layout Eclipse RichClient (KIELER) is a research project about enhancing the graphical model-based design of complex systems [FH09]. A key element is the ability to automatically compute layouts of graphical components within the modeling environment. Enhancements to its predecessor are the integration of a variety of modeling languages and the integration of the project into the rich client platform Eclipse[7]. The project is developed as open source software, licensed under the Eclipse Public License (EPL)[8]. The project is divided into layout, pragmatics, semantics, and Open KIELER as illustrated in Figure 2.3.

Layout concerns the automatic layout of graphical elements. Time-consuming editing of graphs can be saved. Pragmatics concerns the the context of the textual and graphical elements and their relation to the user. This field tries to enhance users experience when using the language(s) of concern. The semantics are concered with the meaning of the used symbols in the languages. This is the part of the project that is mainly concerned with SCCharts.

In the context of this thesis, KIELER is primarily used for visualizing the generated textual SCCharts that result in the implemented visualization. An alternative editor for illustrating SCCharts is KEITH[9].

2.4.4 F-Sharp

F-Sharp (F#) is a functional, fully-typed programming language that also incorporates imperative and object-oriented programming methods. It is a multiparadigm language. The language was developed by Mircrosoft Research in Cambridge and runs on the .NET platform. That means it profits of garbage collection and a broad variety of class libraries. It is a fully

[7] http://www.eclipse.org/

[8] http://rtsys.informatik.uni-kiel.de/~kieler/epl-v10.html

[9] https://rtsys.informatik.uni-kiel.de/confluence/display/KIELER/KEITH

```fsharp
open System // Gets access to functionality in System namespace.

// Defines a function that takes a name and produces a greeting.
let getGreeting name = $"Hello, {name}! Isn't F# great?"

[<EntryPoint>]
let main args =
    // Defines a list of names
    let names = [ "Don"; "Julia"; "Xi" ]

    // Prints a greeting for each name!
    names
    |> List.map getGreeting
    |> List.iter (fun greeting -> printfn $"{greeting}")

    0
```

Figure 2.4. A small example of F-Sharp code. The example is taken from the official Microsoft documentation website.

supported language by IDEs such as Visual Studio Code. In terms of language relations, it is close to Python, C# and Haskell [Smi09].

Introductory guides and information can be found on the official Microsoft .NET documentation site[10]. Figure 2.4 gives a small example of code written in F#.

[10] https://docs.microsoft.com/de-de/dotnet/fsharp/

Chapter 3

Extracting and Generating SCCharts from Blech

The main goal of this thesis is to generate abstract SCCharts from Blech code. The focus lies on the abstraction of Blech code with special interest in visualizing the implicitly contained states inside the code. The intended purpose of the visualization lies in supporting the development process of software. AS explained in the introduction, the currently available visualization of the program graph that visualizes all statements was found to be too confusing and big, as it contained many states that are unnecessary to understand the control flow in the program. Further, that visualization does not focus on the important implicitly contained states of the Blech code. This is an important point to be made, since Blech is a language for reactive, embedded systems. Hence, this thesis does not intend to generate semantically identical SCCharts. Rather we synthesize SCCharts that represent the Blech code in a more abstract matter with focus on the stateful nature of the code. This was discussed in Section 1.1. The development into semantically identical SCCharts is recommended for future work, as explained in detail in Section 6.3.

The first step to take is to determine the desired SCCharts equivalent to individual Blech statements. Translating the statements one by one introduces unneeded hierarchies and transient states through translating non-stateful inner behavior of Blech constructs or inlining activity calls. Hence, to generate the described abstract SCCharts, multiple phases are needed.

The first phase takes care of directly translating Blech code into SCCharts statement by statement. Each statement is translated into an SCCharts element. Blech statements that are not relevant to the stateful nature of the code, such as simple expressions, are neglected in this step, which is compliant to the goal of this thesis. Only elements that are important to the statefulness or corresponding control flow are taken care of. This step is called the structural translation and is covered in Section 3.1.

Next, information is added to the states via labels defined by the user in Blech. This way, for documentation purposes, users might add more information to the graph or they might be able to debug their program better if they understand the context surrounding their defined label by examining the visualization. The extraction of labels is discussed in Section 3.2.

In the last phase, simplification steps are taken to improve the generated SCCharts for a more intuitive and simple illustration of the source code. As discussed in the motivation, unnecessary transient states that are not relevant to the stateful behavior or necessary control flow are not desired in the visualization. Further, we aim for a visualization that is as simple as possible. The simplification steps take care of flattening added hierarchies in the generated SCCharts. Due to the statement by statement translation in the first phase, multiple hierarchies are added, such as a hierarchy for the body of a loop. This is necessary as no information is

```
@[EntryPoint]
activity runningEx
    (in1: int32, in2: bool, in3: bool)
    repeat
        if in1 > 42 then
            await in2
        else then
            cobegin
               await in2
            with
               await in3
            end
        end
    end
end
```

Figure 3.1. Artificial Blech code serving as a running example.

known about the inner behavior of hierarchical elements such as a repeat statements, when encountering the statement. Such hierarchies are to be flattened in the simplification. Further, this phase simplifies the generated SCChart in terms of their immediate transitions. If a state in the SCChart can be exited immediately via an immediate transition, it is not a conceptual real state. Hence, there must be a way to collapse immediate transitions and omit transient states without losing information about real conceptual states. The simplification phase is discussed in Section 3.3 and is divided into flattening of hierarchies and collapsing immedediate transitions and is discussed in detail in Section 3.3.1 and Section 3.3.2, respectively.

Figure 3.1 shows a piece of artificial Blech code. It contains an if-else construct inside a loop. The if branch awaits a boolean input value, whereas the else branch contains a cobegin awaiting a boolean value in each branch.

The code serves as a running example to illustrate the different phases as a whole. The code does not contain all of the possible Blech constructs,else it would be a big complicated visualization and not fitting to illustrate the main ideas of the phases. Showing the visualization of the running example after each phase will make the need for each phase clearer.

3.1 Structural Translation

In the first step, SCCharts are generated from Blech code statement by statement. Since it is desired to illustrate the stateful nature of the code, most statements will be ignored, such as simple calculations. We focus on the statements that switch the implicit state of the code. Generally, the translation sequentially checks the statements of the code and visualizes the

statements accordingly.

Note that different activities in the Blech code might call each other via run statements. It is assumed that a list of activities to visualize is given. This list is handled sequentially.

In the following, the structures and elements of Blech that are to be translated in this section are listed. Additionally, the translation of each of these elements is discussed.

Generally, the translation rules are applied recursively, e. g., the list of statements contained in the body of a loop will be translated as any other list of statements in the program. The body of a hierarchical Blech element will be translated before the rest of the list on the same hierarchy level is evaluated. Unneeded hierarchies originating from this procedure, as well as hierarchies that contain no stateful behavior, will be simplified in the simplification steps following this step.

Moreover, note that some translation rules add an empty state after the states solely concerning the translated Blech construct are added. Adding such state is necessary for serving as a connection point for subsequent statements. This is needed because some translations introduce transitions leaving a complex state. These transitions convey relevant information. Translation rules that just add a transition to the previous available state would break this procedure. Hence, new empty states are added in many translation rules. If this newly added case-closing state is not needed, it will be simplified in the simplification step concerning the deletion of immediate transitions. This includes the case where the referenced activity does not terminate, i. e., does not have a final node (after simplifications).

Note that these translation steps are initial translation steps that do not yield good-looking and compact results yet. However, having simple translation rules makes the translation easier as it allows self-contained translation that does not require look-ahead implementations. As mentioned before, simplification steps are taken to achieve a more readable and compact visualization.

3.1.1 Entry-Point Activity

The outermost part of a visualization is always an activity, such as the activity marked as the entry point in Blech code or a specifically selected activity by the user. In Blech, an *activity* is an enclosed piece of code that contains stateful behavior, e.g. an await statement or a run statement. This differs from *functions* in Blech that do not contain stateful behavior and are used to outsource simple calculations. Consequentially, functions are neglected by this synthesis.

An element specific to activities to be discussed are the in- and output parameters. It was argued that (local) variables and simple expressions are to be neglected as they add no value to the visualization of the stateful behavior of the code. The same could be applied to in- and output variables of activities. However, the interface declared by an activity is very important in Blech. Knowing which variables can be edited inside an activity allows a developer to intuitively make a mental causality analysis if needed. Section 2.4.2 discusses this in detail. Also, the interface gives a clue about the origin of the variables in the visualization, as variables can still be present in conditional edges originating from control flow elements,

as well as conditions on the important edges originating in await statements. If a variable is present that is not part of the activity interface, it can be safely assumed that it is a local variable. If the activity interface were to be left out of the visualization, no clue about the origin of variables would be left. Lastly, the output is catered towards SCCharts and it is possible to hide the in- and output variables of states via configuration. Consequentially, this concept proposes to extract information about the activity interfaces.

An activity is visualized and initialized as follows.

1. Create an SCChart for an activity.
2. The name of the activity is the name of the SCChart.
3. Add input and output variables of the activity as the input and output variables of the SCChart with equal names and types.
4. Add an initial state and a final state.
5. Translate the statement in the body sequentially according to the list given by this section and add them to the body of the generated SCChart. Some Blech elements are illustrated hierarchically, this means that a repeat-loop, for example, is considered to be one statement. Its body is translated hierarchically inside the generated hierarchical state.
6. When all statements are examined, connect the state of the last statement to the final state, except if the last statement was translated to a complex state without a final state such as a repeat without ending conditions or an infinite run statement. If the last statement is such described statement, remove the now unused final state.

An example of the synthesis for an activity is given by Figure 3.2. Figure 3.2a shows an example of Blech code, defining an activity. The code also contains a comment that represents code of the body. It can be of arbitrary complexity. The details of this code are irrelevant for this translation rule. In this and following examples, the abstracted arbitrary code elements are represented by an empty square. Figure 3.2b shows the generated SCChart. The generated SCChart holds the translations rules described above. The abstracted inner behavior is the square that is connected to the states `a` and `b` via edges. The name of the activity is preserved, as well as the input and output variables. The activity contains an initial state `a` and a final state `b`. States, according to the translation of of the body of the activity, were to be illustrated between the initial and final state.

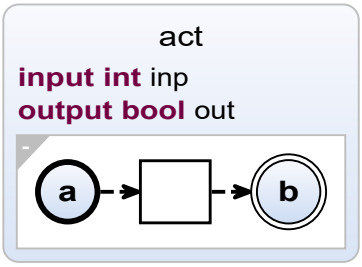
The names of the states `a` and `b` are representative and added for easier description of the figures. Those labels will not be added automatically through the synthesis. However, it is possible to add custom labels for states as will be discussed in Section 3.2. Next, the statements of the body of an activity, or any other body of any construct is illustrated.

3.1.2 Run

As described in the beginning of Section 3.1, each activity and every other hierarchical construct contains at least one state, the initial state. Through step by step translation of the statements, more states are added and connected to the previous states. Therefore, there is

```
activity act(int32 inp)(bool out)
    //...
end
```

(a) An activity in Blech.



(b) The translated SCChart.

Figure 3.2. SCCharts synthesis from Blech activity.

always a previous state available to use as a source of the immediate or delayed transition leading to the state or states introduced for the translation of a certain statement.

The statement introduced in this section is closely related to the previous construct. This subsection covers the run statement. A run statement includes the name of an activity. The run statement starts the execution of an activity. How an activity works and how it itself is illustrated is covered in the previous subsection. A run statement is translated as follows:

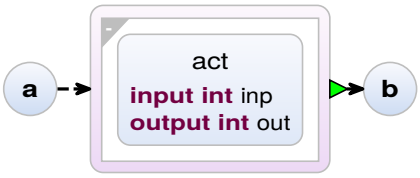
1. Add a new state for the called activity and name it as the name of the activity.
2. Make the state a referenced SCChart that references the SCChart with the name of the called activity.
3. Add an immediate transition from the always present previous state, as mentioned before, to the newly added state.
4. The activity that is referenced is illustrated as its own SCChart as mentioned before, and is of no concern at this point.
5. Connect the complex state via an immediate termination transition to a newly added empty state.

Further, in order to call a referenced SCChart, arguments are needed. If the given arguments in the Blech program are variables rather than literals, the compiler expects variables to be defined in the state that references an SCChart. Therefore it also a necessary step to define the variables that are expected from the referenced SCChart in the calling SCChart. To achieve this, we accumulate all needed variables and their types while synthesizing the body of an activity and add the accumulated variables as local variables at the start of the SCChart. Variables need to be defined at the beginning of a SCChart before any states are specified.

Remember that adding the empty state after the referenced SCChart is necessary for serving as a connection point for subsequent statements. This is needed since a distinct translation of subsequent statements might use different transitions than the terminating transition and the information about the referenced activity needing to terminate to continue the control flow in the graph must be preserved here.

Figure 3.3 shows a translation from a run statement in Blech code as shown in Figure 3.3a into a SCChart as shown in Figure 3.3b. The SCChart shows an empty state `a`. This is the always

```
run act(inp)(out)
```

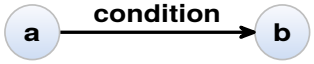


(a) A run statement in Blech.

(b) The translated SCChart.

Figure 3.3. SCCharts synthesis from a run statement in Blech.

```
await condition
```



(a) An await statement in Blech.

(b) The translated SCChart.

Figure 3.4. SCCharts synthesis from an await statement in Blech.

available previous state that was added as a case-closing state from previous statements in the step by step translation or a given initial state that is always present in initial hierarchical elements.. Such state is always available as was discussed in the beginning of this subsection. From such state an immediate termination transition leads to the state that represents the called activity `act`. The name of the activity is preserved, as are its input and output variables. The called activity is a referenced SCChart. This enables the user to expand the referenced SCChart to view the behavior of the called activity. When the activity finished, it connects to a newly added state `c` that serves as a connection point for subsequent statements.

Now that the translation of activities and run statements have been established, their bodies have to be translated as well. The relevant Blech statements are to be translated as follows.

3.1.3 Await

Await in Blech is a simple instruction that holds the execution of a thread and waits for the next tick to await a condition given by this statement. The execution halts until the condition is met. For the implicitly given states in the code, this statement is the most important as it clearly defines the end of a state. A state begins at the end of a previous await statement. However, as explained earlier, the final visualization is more complicated than that, since other statements add different important information to the visualization about the control flow or concurrency, such as if-else statements or cobegin constructs.

To synthesize an await statement add a new state. As described earlier, there is always a previous state available. Connect the previous state that is always present via a delayed transition to the newly added state. Add the condition of the await statement as a trigger to the delayed transition.

Figure 3.4 shows the translation of an await statement to SCCharts. The Blech code given in

Figure 3.4a shows an await statement. The statement itself is simply translated as a transition from the previous state, which is always available as described Section 3.1.2 and denoted as state `a`, to a newly added state `b`. The condition of the await statement is the trigger for the delayed transition.

3.1.4 Repeat

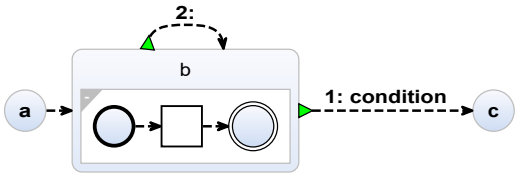
The next Blech element to be discussed is *repeat*. There are two possible repeats, one with a condition that breaks the loop, and an endless loop without such condition.

The translation is simple. The repeat construct is translated hierarchically. The body of the repeat construct will be illustrated in a complex state that is initialized with an initial and a final state that are connected to the other elements of the body as it is the case with an activity and every other hierarchical element. The body will be sequentially synthesized statement by statement. The difference to the hierarchy in activities is the lack of input and output variables. The only information missing now is how to connect the new complex state to its surroundings. As mentioned before, there is always a previous state that the new complex state can be connected to. Therefore, we add an immediate transition from this state to the new complex state. As already mentioned, there are two cases to differ for outgoing transitions of the complex state containing the body. In the first case, there is no end condition. The only case where the control flow leaves the complex body is to reenter it, due to it representing a loop. In this case, a simple immediate termination transition is added with the complex repeat state as source and target. After each execution, the complex state is simply started all over again. The second case covers repeats that do have an ending condition. This case causes the control flow to take one of two ways. The first checks the given condition and if the condition is met, an immediate termination transition will be taken to a new state that serves as the previous state for statements following the repeat. The other way is taken via an immediate termination transition if the condition is not met, and leads back to the complex state representing the body of the repeat, starting the loop all over again. This translation is simple and the introduced hierarchy can be flattened in simplification steps later on.

Figure 3.5 illustrates this translation rule for a repeat construct with an end condition. Figure 3.5a shows a repeat construct with an end condition in Blech code. In Figure 3.5b, the previous state `a` leads with an immediate transition to a state with two outgoing transitions. That state is denoted as state `b` and is the complex state and would contain the illustration of the body of the repeat construct, which is abstractly illustrated between the initial and final state of the complex state. Two outgoing termination transitions show the control flow of the end condition of the repeat construct. If the condition is met, a transition leads to state `c` and if not, the other transition leads back to the complex state itself. State `c` is the state subsequent statements are connected to.

A repeat construct without an end condition is not visualized here because of the triviality of the translation. If the example shown in Figure 3.5 were a repeat construct without an end condition, the transition with the end condition and its target would disappear.

```
repeat
    // ...
until condition end
```



(a) A repeat construct with an end condition.

(b) The translated SCChart.

Figure 3.5. SCCharts synthesis from repeat construct in Blech.

3.1.5 While

The while construct is very similar to the repeat construct described before. The main difference in the workflow is that the condition is checked at the beginning of the loop. The loop-starting state, which was considered to be the previous state in previous translation steps, therefore has a transition into a complex state containing the body of the loop and to a state that was newly added and is the starting point for connecting subsequent statements. If the previous state is simple, it simply serves as a connection point for transitions and is not really a state. If this is the case, the previous node can be replaced by a connector state, a special state in SCCharts that serves exactly this purpose. The body of the while loop is illustrated according to the list this section gives and starts with an initial state and a final state, which are connected to the rest of the states in the same manner as was the case in activities and repeats. Further, an immediate termination transition is added that leads from the complex state to the state starting the loop.

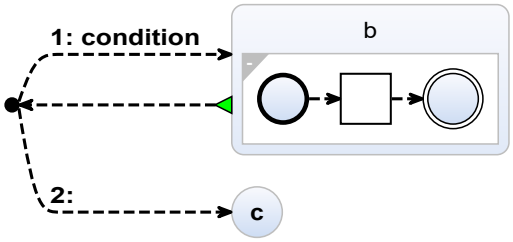
Figure 3.6 illustrates such translation. Figure 3.6a shows a Blech code example with a while-loop that is ended through a condition. Using the above translation, this Blech code translates to the SCChart illustrated in Figure 3.6b. The previous state, which is always present as discussed before, connects to two states. As described above, it has been transformed into a connector state. If a condition is met, it connects to state `b` that would contain the body of the while loop. The body would be illustrated between the initial and final state of the complex state is abstractly illustrated. If the condition is not met, the connector state connects to state `c`, which would be the connection point for subsequent statements.

3.1.6 Cobegin

Cobegin is a construct in Blech that represents concurrency. The statements inside the different branches of a cobegin construct are executed concurrently. Branches can also be weak, which means they are weakly aborted by other branches that have terminated. For example, an endless loop in a weak branch can be aborted by a simple await statement in another branch. The await branch terminates on a given condition and then aborts the weak branch. When aborted, a weak branch will continue to execute until it is paused (for example by an await statement). We illustrate a cobegin construct as a complex state. We connect such state with the previous state with an immediate transition. To illustrate the concurrency, SCCharts offer

```
while condition repeat
    // ...
end
```

(a) A while construct with an end condition in Blech.



(b) The translated SCChart. The source of the conditional transitions, state a, has been transformed into a connector state.

Figure 3.6. SCCharts synthesis from Blech construct while.

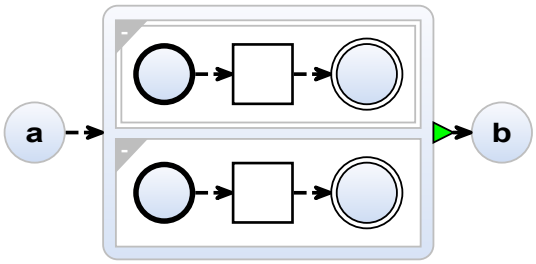
multiple regions in states. Regions are also executed concurrently in SCCharts. Hence, a cobegin construct is illustrated as a state with a region for every branch of the cobegin. Every region, much like an activity as described in Section 3.1.1, is initialized with an initial state and a final state. From there the bodies of the branches are illustrated in the corresponding region according to the instructions given by this section. The final state is connected, and possibly deleted, according to the same rules that were described in Section 3.1.1. That means a termination transition is added, if the the cobegin can terminate. If this is the case, the complex state illustrating the inner behavior of the branches leads to a newly added state with an immediate termination transition. This new state serves as the starting point for subsequent statements.

Illustrating weak branches is achieved by making the regions representing the weak branches *final*. Making a region final adds a visual clue to the illustration in form of a second rectangle surrounding the region. This way, the user knows which branches are weak. In SCCharts, final regions mark every state of the region as final. In the current implementation of SCCharts, if all regions are weak, the super state can be exited immediately. Therefore, a small change is required so that any of the weak regions have to terminate first for the super state to be exitable. With this change, multiple weak branches execute until a branch is terminated.

The Blech code in Figure 3.7a shows a a cobegin construct with a weak and a normal branch. Figure 3.7b shows the corresponding SCChart. A transition leads from a previous state denoted as state a to a state with two regions, that represents the cobegin structure with two branches. The regions, as all hierarchical elements before, contain an initial and a final state that are connected to the rest of the bodies of the branches as usual. The first branch is denoted as weak in the code and thus, it is marked as final in the SCChart. The complex state, as the translation dictates, leads with an immediate termination transition to a new state, denoted as state b.

```
cobegin weak
    // ..
with
    // ..
end
```

(a) A cobegin construct with a weak and a normal branch in Blech.



(b) The translated SCChart.

Figure 3.7. SCCharts synthesis from Blech construct cobegin.

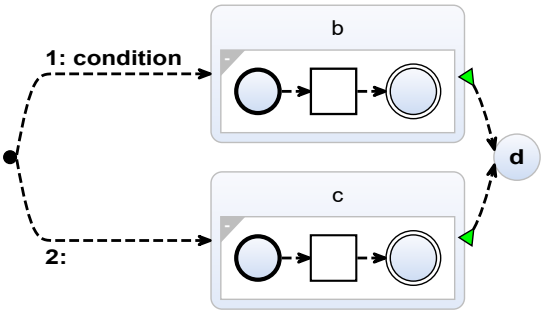
3.1.7 If-else

If-else is a common construct in many programming languages. It is not different in the Blech language. Depending on the condition, a branch of execution is selected. To translate such construct into SCCharts, we add one state for each case of the if-else construct and a closing state that connects all the different branches back together. For example, an if-else construct with two else-cases would add three states for the branches and one closing state. Overall, four states are added in this example. The previous state that is available in every case as was discussed before is the state we connect the states for each branch to. The previous state serves as the source of the immediate transitions. The respective branch states are the targets. If the previous state is simple, it simply serves as a connection point for transitions and is not really a state. If this is the case, the previous node can be replaced by a connector state, a special state in SCCharts that serves exactly this purpose. The condition of the if or else-if branches are added to their respective immediate transition. The state for each branch is a complex state whose inner behavior is the body of the corresponding branch of the if-else construct. These complex states are synthesized as usual with an initial and final state that are connected to the rest of the body as described before. The complex states then connect with an immediate termination transition to the already described case-closing state. That state serves as the connection point for subsequent statements. The connection to the case-closing state only happens for branches where the complex state contains a final state.

An example of this is illustrated in Figure 3.8. The Blech code in Figure 3.8a shows a simple if-else construct. This is translated into the SCChart shown in Figure 3.8b, where each branch has a transition leading to it, according to the conditions given by the construct. The always present previous state has been replaced by a connector state. State `b` is reached via a transition with a condition as was set in the Blech code. State `c` is the else case. Their inner behavior is shown abstractly. State `d` is the closing state, reconnecting the branches and serving as a connection point for subsequent transitions.

```
if condition
    // ..
else
    // ..
end
```

(a) An if-else construct in Blech.

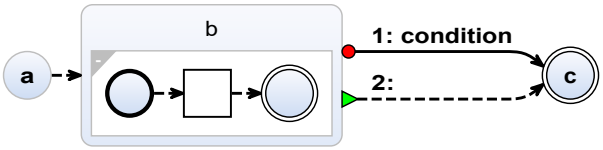


(b) The translated SCChart. The previous state is transformed into a connector state.

Figure 3.8. SCCharts synthesis from Blech construct if-else.

```
when condition abort
   // ..
end
```

(a) A when-abort construct in Blech.



(b) The translated SCChart.

Figure 3.9. SCCharts synthesis from Blech construct when-abort.

3.1.8 When-abort

The next construct to be discussed is the when-abort construct. The inner behavior of the body is aborted when a specific condition is met, but only after the execution in the body was halted for the first time. After that, the condition is checked at the beginning of each reaction.

The body of the when-abort is nested inside a complex state, which is as usual initialized with an initial state and final state. The abort condition given by the when-abort construct is added as an abort transition leading from the complex state to a newly added state that closes the abort and regular termination cases. If the body of the when-abort construct terminates by itself (has a final state after synthesizing the body), an immediate terminating transition without a trigger is added leading from the complex to the new state.

This translation is illustrated in Figure 3.9, where Figure 3.9a shows Blech code with a simple when-abort construct. This is translated as described above to the SCChart shown in Figure 3.9b. State `a` is the previous state that is always available. It leads to state `b` with an immediate transition. State `b` is the state that contains the body of the when-abort construct, which is abstractly visualized here. It leads with an abort transition and immediate termination transition to state `c`, which is the closing state and serves as a conenction point for subsequent statements.

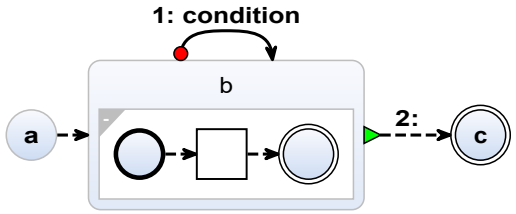
3.1.9 When-reset

Lastly discussed is the when-reset construct. It is very similar to the when-abort introduced in Section 3.1.8. This construct does not abort and then skip the body when the condition is met, rather it goes back to the beginning of the body and tries to re-execute it.

This difference is used in this translation. The translation is the same as in Section 3.1.8, except for the abort transition: it targets the complex when-abort state as the abortion reenters the execution of the complex body. This is shown in Figure 3.10, where the abort transition does not target the following state but rather the same complex state again.

```
when condition reset
   // ...
end
```

(a) A when-reset construct in Blech.



(b) The translated SCChart.

Figure 3.10. SCCharts synthesis from Blech construct when-reset.

Running example after the translation step

Figure 3.11 shows the visualization of the running example after the initial translation step. The single statements have been transformed into their set SCChart counterparts. The bodies of hierarchical constructs are contained inside complex states. The activity name and the parameters were preserved. Await statements were realized with delayed transitions.

The need for the simplification steps can be observed by this example. The hierarchy on the if-else construct as well as the self-loop, which was caused by the Blech repeat construct, is not necessary for showing the stateful nature of the code. They fill the visualization with unnecessary elements. Hence, the are to be removed in the following simplification step. The need to eliminate transient states can be observed in this example as well. Every hierarchy contains a state that is connected via immediate transition to the final state. Such transient states do not convey information as they can be exited immediately and are to be removed as well.

3.2 Label Extraction

Using only the translations described in Section 3.1, most states are empty in terms of their label. They do not have any labels further describing them. The labels that are shown in the figures in Section 3.1 are placeholder labels that are placed intentionally for easier descriptions of the illustrations. The goal of this thesis, however, is to give the developer of

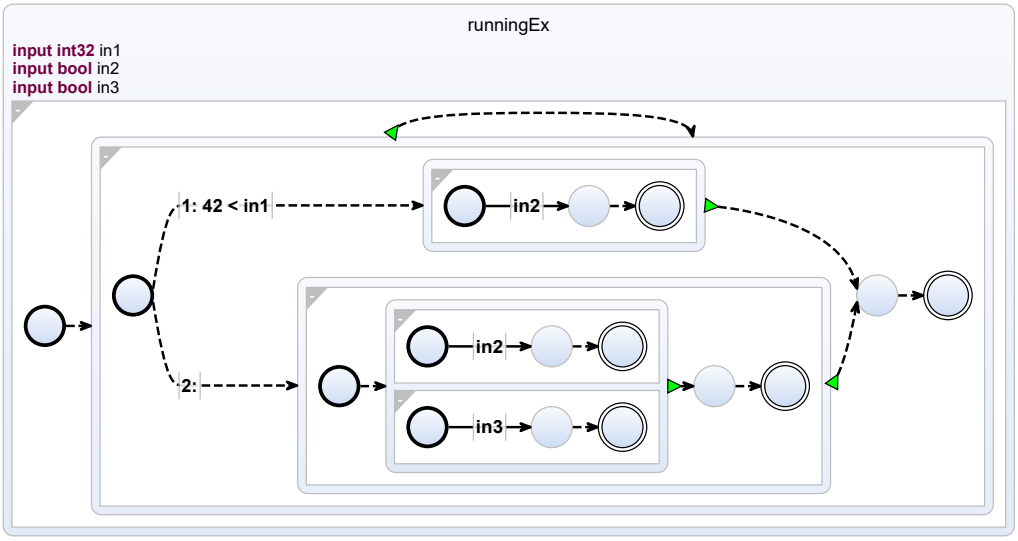


Figure 3.11. Visualization of the running example given by Figure 3.1 after the initial translation phase.

the Blech code an abstract overview, showing the stateful nature of the code and enabling the usage of the illustration for documentation purposes. If the states are supposed to have documentary character, they need to be labeled. This reasoning was already discussed in Chapter 1. Enabling labeling, the states will have a distinct developer-chosen description through a given name. The developer could enhance the visual representation of the code. This is especially helpful if the visualization is used in documentation contexts. The labels for states can be used to highlight implicit information hidden in the code and enhance the recognition of the developers mental map.

This thesis proposes two approaches to labeling the abstract illustrations: State only and an extended version.

The first approach focuses solely on actual conceptual states, which are implicitly given by await statements notifying about the state changes. In theory, transient states are collapsed so that only the states prior to await transitions remain. These are the states that are to be labeled in the first approach. The way this labeling works is discussed in Section 3.2.1.

The second approach is more flexible. After the state-only labeling, only cobegin constructs and their regions remain to be labeled. In the simplified version of the illustration, after the simplification steps of Section 3.3 have taken place, only cobegin constructs and referenced SCCharts via run statements in Blech remain as hierarchical elements as all other have been flattened. Complex states originating from run statements are already labeled, as they are given the name of the activity they call/represent. That leaves the states and regions caused by cobegin constructs to be labeled. This second approach is discussed in Section 3.2.2.

To keep consistency with common programming languages, the labels (comments) are specified before the constructs of interest. A prominent comparable example are Javadocs

```
@@[label="aLabel"]
await condition
```

(a) Await in Blech with label specifications.

(b) The translated SCChart.

Figure 3.12. Label extraction from specified labels for an await statement in Blech code.

in the Java programming language[1]. The specified labels are realized via pragmas in this concept. In Blech, they are realized via a double @ notation. Altogether the labels are specified as *@@[keyword="placeholder"]*, where keyword describes the element to be labeled and the text *placeholder* in between the quotation mark is a placeholder for the sought label.

3.2.1 State-only

The await construct adds a new state and a transition. The transition is labeled with the condition of the await statement. When labeling an await statement, a user may want to describe the state the program is in when the program awaits a certain condition. Therefore, the state that comes before the transition must be the one that is labeled. This is always possible, because there is always a previous state available, either through initialization (initial state) or with the help of the translation rules given by Section 3.1. All of the translation rules add an empty state that concludes the translation of a statement or construct. Therefore, labeling an empty state without interfering with other labeled states (e.g. run statements) is possible.

The labels for awaits are specified via the annotation *@@[label="text"]*, where *text* is a placeholder for the desired label to be assigned to a state.

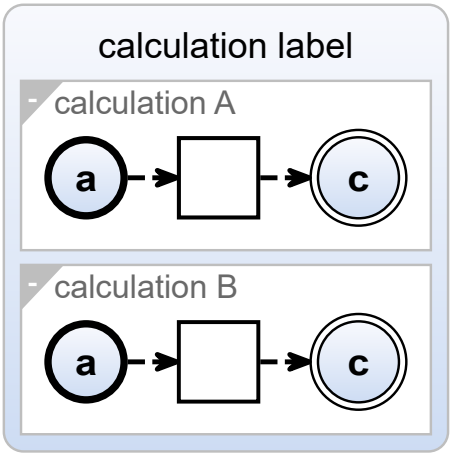
Such proposed label specification and placement is shown in Figure 3.12. Figure 3.12a shows a specified label for an await statement. Figure 3.12b shows the generated SCChart with the specified label. The label *aLabel* is placed in the state before the immediate transition as described above.

There are further considerations to be made for this approach. One is the scenario of multiple specified labels for an await statement, e.g. a user labels a set of simple expressions calculating some value and then adds another label before a following await statement. It would be feasible to ignore all duplicate labels and labels that cannot be directly assigned to an await statement and only consider a label if it is specified right before a statement. Alternatively, it is possible to consider all specified labels, merge them if necessary and assign them to their corresponding real state. This thesis proposes the latter approach. In the example given at the beginning of this paragraph, the labels would be merged in the state that is the source of the awaiting transition. Keeping everything that the programmer specified and not throwing anything out without giving the programmer a notice maintains the programmer's control of the visualization. This way, everything the programmer does is preserved and there

[1] https://docs.oracle.com/javase/8/docs/technotes/guides/language/index.html

```
@@[cobegin="caluclation label"]
@@[branch= "calculation A"]
@@[branch= "calculation B"]
cobegin
   // ...
with
   // ...
end
```

(a) Cobegin in Blech with extended label specifications.



(b) The translated SCChart.

Figure 3.13. Label extraction from specified labels for cobegin constructs in Blech code.

is no need for the programmer to know the exact syntax and semantics of specifying a label. Another point to consider is cases where the program is non-linear, e.g., if-else constructs. A possible scenario to consider would be a specified label right before an if-else construct with both branches only containing an await statement. The issue here is that it is not clear how to assign the label. In the first of the above described two options, the label would simply be discarded, but since this thesis proposes the second alternative due to the above mentioned reasons, the label would be duplicated for both if-else branches and assigned for both await constructs. This is feasible, because the two then labeled states are the same logical state.

3.2.2 Advanced approach

The extended approach allows for a more fine granular labeling of the elements. After the simplification steps following later, the only hierarchies remaining are due to run statements or cobegin constructs. Since run statement-caused states already have a label given by the activity they are referencing, they are of no concern. That leaves the labeling for cobegin-caused visual elements. The elements to be labeled are different regions, presenting a cobegin branch each, and the complex node containing these regions. For the extended approach, more labeling keywords are introduced to distinctly assign the labels and avoid assignment issues. The keywords cobegin and branch are proposed. The labels are then assigned via *@@[cobegin="text"]* for the complex node and *@@[branch="text"]* for a branch. All labels specified in the Blech code for the visualization are to be placed before the cobegin construct to not cause assignment issues and the need to backtrack from inside a branch-body back to the outside. Consequentially, a branch label is needed for each branch, when labeling the different regions, in order to make the labeling unambiguous.

Such extended label specification and placement is shown in Figure 3.13. Figure 3.13a shows the Blech code containing a cobegin construct and the above mentioned label specifica-

tions, one for each branch of the cobegin. Their visualization is shown in Figure 3.13b. The complex state and regions are labeled according to the specified labels and above described label placement.

3.3 Transient State Elimination & Hierarchy Flattening

Now that simple SCCharts were generated, we want to take a couple simplification steps to increase further readability and simplicity of the resulting graphs.

This concept proposes two simplification steps. First, flattening of hierarchy. As described in Section 3.1, most translation steps introduce complex states with inner behavior. Those states add unnecessary hierarchy to the illustration, which further increases the complexity. Section 1.1 discussed that a simplistic illustration that helps developers understand the code quickly is the target to be achieved. Complexity may be a hindering factor to that. Hence, we want to reduce the complexity of the illustration by flattening the hierarchy as much as possible. This is discussed in Section 3.3.1. Further, it is found that not all hierarchies can be flattened.

The second simplification step targets to collapse immediate transitions. They cause transient states and do not enhance the ability to understand the stateful nature of the code. Transient states are exited immediately due to the immediate transitions. Hence, transient states are not real states. Therefore, collapsing immediate transitions and transient states is an important simplification step for a better result. This step follows after the flattening hierarchies step because flattening hierarchies produces more immediate transitions that can be collapsed in the second simplification step.

3.3.1 Flatten Hierarchy

The flattening of hierarchy is the first simplification step. As will be shown, immediate transitions are introduced in this step. Hence, it must come before the elimination of transient states.

The constructs that add complex state are: run, repeat, while, cobegin, if-else, when-abort, and when-reset. The constructs activity, repeat, while, if-else form a group of similar constructs and are discussed together. They are similar because they have complex states with simple incoming and outgoing transitions. When-abort and when-reset also form a similar group. They are similar because they have an abort-transition as was described before and possibly have a second transition for regular termination. Cobegin is a special case because it might contain multiple regions and is therefore generally hard to flatten.

It is desired to make the flattening of these hierarchies configurable. This is especially important on different alternatives as introduced for the flattening of cobegin constructs as will be discussed later.

Note that flattening hierarchy and removing complex states has implications on the labels described in Section 3.2. The only named complex states are ones caused by run statements

or cobegin constructs. Their labels will be discarded when flattening their hierarchy. The complexity is what is labeled. If the programmers label something that they want removed through configuration, it can be expected that removed labeled elements will not be found in the illustration. Labels resulting from await statements will not be affected in this step. Since these labels are always set on simple states as discussed in Section 3.2.1, their labels only matter when collapsing immediate transitions, after hierarchy was flattened.

Run, Repeat, While & If-else

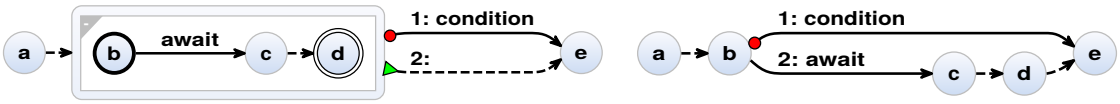
Flattening the hierarchy for these constructs is straightforward. Throw away the complex state and lift the inner SCChart one hierarchy level up. Then connect the incoming transitions to the initial state of the inner SCChart. If the inner behavior has a final state, the complex state has an outgoing transition. Change the source of the outgoing transitions from the complex state to the uplifted final state of the inner behavior. The outgoing transitions are changed from termination to a regular transitions. The final and initial states of the inner graph lose their special status and become regular states. Note that self-looping transitions have both their target and source switched. Furthermore, note that information about the called activity label and in- and output variable names are lost when flattening a run statement caused hierarchy.

The final inner node of the complex state in repeat contexts is special, because it is the origin for two conditional transitions (repeat the loop and stop the loop). As was the case in Section 3.1.5 and Section 3.1.7, such state can be transformed into a connector state.

In this group, hierarchies resulting from run statements are also special, since the called activities are their own self-contained pieces of code and the programmer intended to separate them. Breaking hierarchy for complex states that are derived from activities by default is not always desired. As activities represent self-contained parts of the source code, one could rather just inspect the generated chart for said code, activity per activity. This concept recommends to generally keep the hierarchy caused by run statements, because outsourcing code into activities is separation and thus, intended hierarchy. But sometimes, especially for small repetitive activities, it might be desired to break the hierarchy nonetheless. Therefore, due to the special nature of run statements, a configuration for the flattening of run statement caused hierarchies could be beneficial.

When-abort & When-reset

Flattening the hierarchy for this group is also easy but different to the group discussed before due to the abort transitions. The inner behavior of the construct is nested inside the complex state. Flattening the hierarchy has following implications: The super state disappears, the inner state chart is extracted to the higher hierarchy level. The initial state loses initial state status. The transition or transitions coming into the previous complex state have their target changed to the former initial state of the inner behavior. The final state of the complex state becomes a regular state and is now the source of the termination transition that represents the regular termination (if present, compare Section 3.1.8), as the complex state disappears.



(a) When-abort construct before flattening.

(b) When-abort construct after flattening.

Figure 3.14. An example of how hierarchy is broken for the when-abort construct.

Further, the termination transition is transformed to a regular immediate transition. Until now the flattening is equal to the previous flattening presented in Section 3.3.1. Next, the abort transition is duplicated. It is needed multiple times, depending on the inner behavior of the construct. Next, update the source of the duplicated transitions in such way, that each state that is the source of a delayed transitions originating from an await statement or complex states originating from a run statement are the source of an abort transition. The target of the abort transition is changed to the former initial state for when-reset constructs and the construct closing state as introduced in Section 3.1.8 for when-abort constructs. As a reminder, the case closing state is state `c` in the translation example of when-abort in Figure 3.9. The limitation of abort transitions having a source in states that are awaiting states or running states is caused by the semantics of then when-abort/when-reset constructs in Blech. The body will execute until the system is halted for the first time. Then, if the program is resumed, the condition is checked and if it is met, the body is aborted.

Note that this collapse of hierarchy causes complexity in terms of adding multiple transitions. If the when-abort construct were to contain a more complex chart, the number of added transitions would increase. This might further add to the complexity of the illustration rather than simplify it. Being able to configure the flattening step for this construct would be beneficial for the developer using the illustration.

The flattening for a when-abort construct is shown in Figure 3.14. Figure 3.14a shows the chart before, Figure 3.14b after the flattening step. Figure 3.14a shows a when-abort statement that contains a single await statement. A boolean value *condition* is the condition for abortion. The flattening step causes unnecessary states such as the final state (state `d` in the example) of complex states to be preserved. In Figure 3.14b, all states are preserved, only their final or initial status was changed to a regular status. The initial immediate transition points to the former initial state `b` of the complex state, and the regular termination transition now is a regular immediate transition and has the former final state `d` of the complex state as source. The abort transition has its source changed to `b`, which is the sole awaiting state of the inner behavior, and still points to the case closing state `e`. Unnecessary states such as `c` and `d` will be discarded in the simplification step of collapsing immediate transitions.

Cobegin & Weak-Abort pattern

Cobegin hierarchy is similar to an activity because it is not feasible break the hierarchy. The cobegin construct is special because it represents concurrency. The translated SCChart has multiple regions. If we were to break the hierarchy, we would loose the regions and thus

any illustration of concurrency. Illustrating the concurrency is important because the final illustration is supposed to help the developers understand the code as described in Section 1.1. Therefore, removing the visual key about concurrency is not desired. However, as was the case with activities, there are patterns that can be identified and thus, simplified.

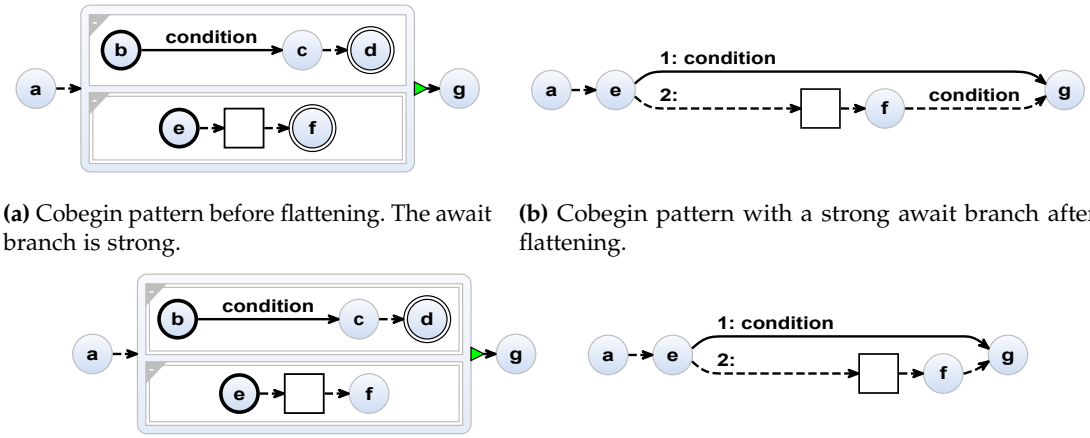
A common pattern is to use cobegin constructs to model weak-aborts, where one of one or more branches contains only a single await statement where the other branches are weak branches and have some stateful behavior inside. The weak-branches will be weakly aborted when the branch containing only the await statement terminates. Hence, the condition in the branch with the await statement is the condition weak-aborting the other branches.

This pattern, however, requires exactly two branches. The branch containing the await statement can be both, weak or strong and is called the *awaiting-branch* in the following paragraphs. When the condition of the awaiting branch is met, that branch will terminate and thus, terminate the other branch because of its weak property. So the weak branch is terminated, we can say aborted, by the condition that is awaited in the non-weak branch.

Thus, a possible simplification is to achieve an illustration for this pattern similar to the illustration of a hierarchy broken when-abort construct. To show the semantical differences above, an immediate transition is used instead of an abort transition. The abort condition is, as the pattern describes, the condition of the await statement of the non-weak branch. That branch containing only the await statement is erased and interpreted as an abort condition for the remaining branch with an immediate transition as the abort transition. We can then flatten the hierarchy the way when-abort constructs were flattened. What is left is similar to a flattened when-abort construct. There are two key differences, however.

The first key difference is a differentiated consideration of the very first weak-aborting transition. The first time the awaiting branch can abort the other non-awaiting branch is at the end of the first reaction. That is the very first time the non-awaiting branch is halted by a delayed transition or a run-statement caused complex state on every path from initial to final state in the branch. Activities must have stateful behavior in Blech, that is why they are considered to be of halting behavior. If an immediate transition were used for the weak-abortion, the execution in SCCharts would take that path immediately. This is not the desired behavior, so the very first weak-abort transition needs to be a delayed transition. However, this is a semantic problem, as the aborting signal can be present in the reaction in which the awaiting branch terminates and absent in the reaction after. The flattened SCChart with the initial delayed reaction would semantically expect the aborting condition one reaction too late. Hence, this pattern is not recommended to use when trying to achieve a semantically correct visualization.

The other difference is the transition from the final state of the inner behavior of the non-awaiting branch. The regular connection via immediate transition from final state to the case-closing state is not realized. If the awaiting-branch is strong, the await condition has to be met to terminate the whole cobegin construct. So an immediate transition with the awaiting condition is added that can then exit the behavior of the non-awaiting branch, which complies with the behavior of the Blech code. If the awaiting-branch is weak, it can be aborted



(a) Cobegin pattern before flattening. The await branch is strong.

(b) Cobegin pattern with a strong await branch after flattening.

(c) Cobegin pattern before flattening. The await branch is weak.

(d) Cobegin pattern with a weak await branch after flattening.

Figure 3.15. An example of how hierarchy is broken for a cobegin pattern. Note that through the abstraction in Figure 3.15b and Figure 3.15d, aborting edges originating inside the black square are left out as well.

by the non-awaiting branch, so the cobegin construct terminates, if the non-awaiting branch finishes. Thus, if the awaiting branch is weak, only a simple regular immediate transition is added from the final state of the inner behavior to the case-closing state.

Figure 3.15 shows such a pattern before flattening. Figure 3.15a and Figure 3.15c are the charts before, Figure 3.15b and Figure 3.15d after the flattening of the hierarchy. Figure 3.15a and Figure 3.15c show the pattern, with an awaiting branch and the other branch being a weak branch. In Figure 3.15a the await statement is inside a strong branch, whereas Figure 3.15c shows the await statement in a weak branch. The condition of the await statement becomes the condition for the abort transitions in the flattened versions. The abort transitions are delayed transitions because they are added to the very first awaiting node e. Following aborting transitions would be immediate transitions. The branch not containing the await statement is weak and contains some states that are arbitrary and abstractly illustrated here. The region matching the await-pattern will be discarded completely, the rest is analog as to the simplification of a when-abort construct when considering the await condition to be the abortion condition, which leads to the simplified illustration of Figure 3.15b and Figure 3.15d. The difference between the two is the different transition from the former final state to the case closing state. The condition of the await statement has to be met to terminate the whole cobegin construct in Figure 3.15b, whereas in Figure 3.15d, an immediate transition without a trigger is used. Note that the abstracted graph inside the black square could, depending on the content, be the origin of immediate transitions with the identified aborting condition connecting to state g.

An alternative simplified visualization of the detected weak-abort pattern is given as follows. This version is independent og the amount of branches present. The pattern only

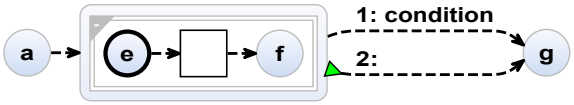


Figure 3.16. Alternative weak-abort pattern visualization to Figure 3.15a.

requires one awaiting-branch and that all the other non-awaiting branches are weak branches. This version does not aim to eliminate the hierarchy, but tries to get rid of the awaiting branch. The awaiting branch gives the aborting condition. Once extracted, the whole branch and region can be discarded. The rest of the complex state can remain as well as the outgoing termination transition, if present. The weak-abortion is realized with an immediate transition having the aborting condition as a trigger. Thus, the awaiting branch was simplified to a transition and the complex state was made less complex. This alternative is closer to the semantic equivalence and achieves to simplify the identified pattern.

Figure 3.16 shows the alternative visualization, to the non-flattened version given by Figure 3.15a. The condition given by the awaiting-branch is added as a weak-aborting transition the the complex state. The non-awaiting region in the complex state remains unchanged. If multiple regions were present besides the awaiting branch, they would be unchanged as well.

With the two alternative versions of simplifying the detected weak-abort pattern as well as the option to not simplify the pattern at all, it results in three versions, each with their own advantages. To give the users more flexibility and the opportunity to use the strengths of each version, it is beneficial to freely allow the configuration between these variants as is shown in the evaluation later.

Running example after flattening of hierarchy

Figure 3.17 shows the visualization of the running example after the hierarchy flattening step. The hierarchies of the if-else and repeat constructs are flattened. Incoming edges changed their targets to the former initial states, whereas outgoing edges changed their source to the former final states. The hierarchy of the cobegin construct was not flattened, since it does not match the described cobegin pattern.

The need for further simplifications is obvious. There are many transient states. Some of them have been introduced by the flattening step.

3.3.2 Transient State Elimination

Flattening immediate transitions and removing transient states is sought in order to keep only real states as good as possible, especially with the translation steps and hierarchy flattening measures introduced in this thesis. To connect states properly, multiple helper states were introduced in the translation steps. Most of these are not to be kept in the final illustration showing the conceptual states as they are transient states. Transient states are

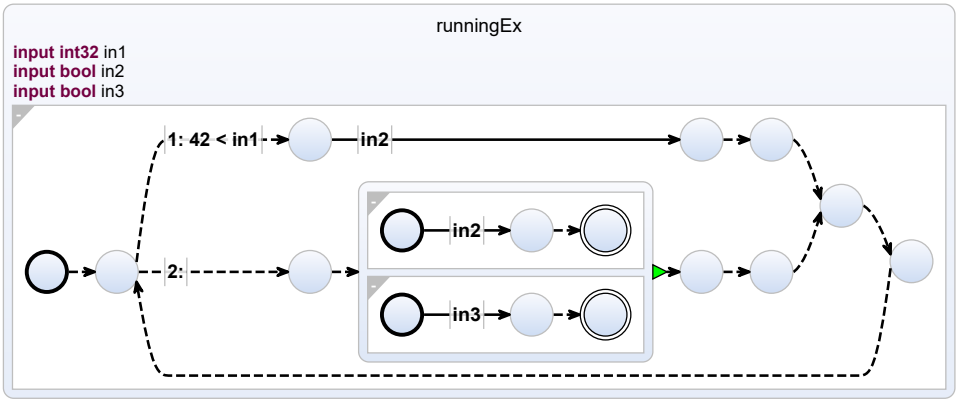


Figure 3.17. Visualization of the running example given by Figure 3.1 after the flattening of hierarchy.

states connected with immediate transitions. The immediate transitions cause the states to be exited immediately. Therefore, they are not real conceptual states of a program.

The challenge is to maintain the correct use of immediate termination transitions. They are used for exiting complex states as stated earlier. Some of them are transformed during the flattening of hierarchies as described in Section 3.3.1, but not all. Most cobegin constructs will not match the described pattern for flattening and keep their hierarchy. Plus, depending on configuration settings specifying that some degree of flattening is not desired, more termination transitions are present. Their presence has to be considered when proposing the collapse of transitions.

Since termination transitions can also be immediate, we can ignore their terminating property and treat them as normal immediate transitions for this simplification step. When reassigning the source of an edge, depending on the new source, the newly connected transitions are simply transformed into terminal or regular transitions, as given.

The general idea is to remove as many immediate transitions as possible. This concept proposes to remove the immediate transitions as follows: If there is only one or many immediate transitions between the source and the target, we first check the source and then the target of the transition. If the source is not a complex state, we simply delete the immediate transitions and the source state. Transitions that originally had the deleted source state as a target then update their target to the target state of the deleted transition. If the source state is a complex state, we check the target state instead and if it is a simple state, we delete it and the corresponding transitions. Analogously, transitions that had the deleted state as a source now update their source to the source of the deleted transition. An exception to this rule is single conditional transitions (with a trigger) as they are important to the control flow of the graph. If there are multiple conditional transitions between source and target, they can be deleted, since this can only result in a simplified if-else case that had no stateful behavior and has already been simplified. Another exception is given by the alternative representation of the weak-abort pattern in cobegin constructs as described in Section 3.3.1. The complex state will have an aborting immediate transition and a terminal transition as outgoing transitions. These

will not be deleted, as the important information of weak-aborting the cobegin construct would be lost.

Note that if labeled states are deleted, their label will be appended to the end of label of the source, if a target is deleted, and to the front of the label of the target, if a source is deleted.

Further, if a checked source and target state have one or many (non-terminating) immediate transitions plus an abort transition between them, these transitions will also be deleted, as they will not have impact anyway due to the immediate transitions. This addition is important for simplifying the illustration after hierarchy was broken from when-abort and when-reset constructs as described in Section 3.3.1. The deletions for this rule follow the same principle as in the previous paragraph, a source or target needs to be simple for this to work. If either is simple, the node in question is deleted alongside the transitions in question, and in- or outgoing transitions are assigned accordingly.

Note that transitions that are assigned to a new source change their status to termination if the new source is complex (and has a final node).

Next case to consider are immediate termination transitions that originate in complex states along with abort or delayed transitions or on their own. If these complex states, such as ones caused by run statements, do not have a final node (i. e., are non-terminating), the immediate termination transition is removed, as it cannot fire.

There are further exceptions to these rules. First, if the state that is to be deleted is an initial or final state, and has more than one outgoing and incoming transition respectively, the state and transition will also not be deleted. This is because it is difficult to reassign the final and initial state status if there are multiple transitions available and the initial state needs to be distinct in SCCharts. It is possible to do for the final state, but this concept proposes not to, in order to distinctly provide a visual ending point. If the transition is a single transition between source and target, the next state for initial states and the previous states for final states are assigned the respective statuses. Note that it is possible for a node to be both, an initial and final node. A scenario where this can happen is an activity that simply calls another activity. The immediate transitions are collapsed and the final and initial status reassigned to the complex state referencing the called activity.

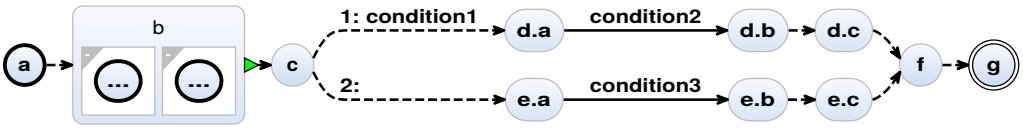
States that are often affected by this simplification step are the initial and final state of complex states, as well as states that were added after certain constructs to serve as a connection point to subsequent statements.

An example of immediate transition simplification is given in Figure 3.18. Figure 3.18b and Figure 3.18c show an illustration of Blech code before and after immediate transition simplification, respectively. Both examples have already been flattened in terms of hierarchy. Figure 3.18a shows the corresponding Blech code. The Blech code can be nested inside any hierarchy, such as cobegin-caused regions or inside an activity. Note that in this example, the content of the complex state has been hidden to focus on the collapse of immediate transitions. Further, note that this is a purely artificial code example.

Figure 3.18b shows the illustration of Blech code that contains a cobegin without weak

```
cobegin
   //...
with
   //...
end
if condition1 then
   await condition2
else
   await condition3
end
```

(a) Example code for a illustration of collapsing immediate transitions. This code can be nested inside an activity or a cobegin region.



(b) Immediate transitions before simplification.

(c) Immediate transitions after simplification.

Figure 3.18. An example of how immediate transitions are simplified.

branches and with unknown content, followed by an if-else construct containing an await statement each. States a and g are the states that are initially added through the activity hierarchy the code is placed in. State b and c are caused by the cobegin, where state b contains the branches and their content, and state c is the case closing state and connection point for subsequent statements as described in Section 3.1.6. From state c the branches of an if-else construct exit into two branches gated by a boolean condition *condition*. The states d.a, d.b, d.c, e.a, e.b and e.c are the flattened bodies of if-else construct. Both bodies contain a simple await statetment with the trigger *condition2*. The control flow is closed again by state f as was described in Section 3.1.7. According to Section 3.1.1, state f then connects to the final state g.

Figure 3.18c shows the simplified version of Figure 3.18b. The states a, c, d.b, d.c, e.b, e.c and f have been discarded due to the simplification. The transition between states a and b was deleted because the source a is a simple state. Hence, a was discarded as well. Through this deletion, state b is transformed to an initial state. The state c and the transition between b and c is deleted because state c is a simple state. Consequentially, the transitions between c the contents of the if-else bodies change their source from c to b. Since b is a complex state, the transitions are transformed to terminal transitions. The conditional and delayed transitions

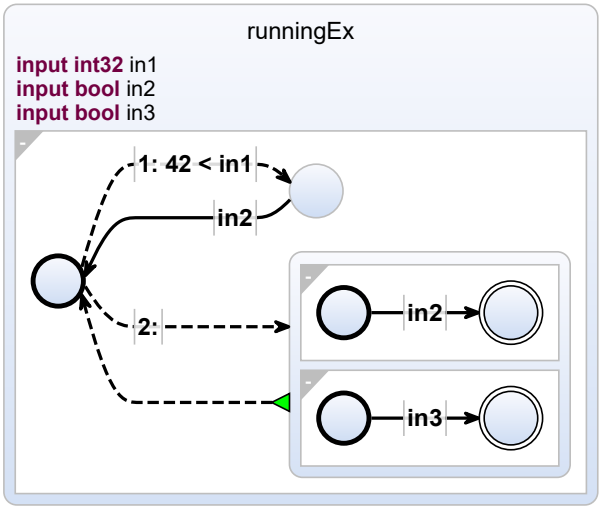


Figure 3.19. Visualization of the running example given by Figure 3.1 after eliminating transient states.

are not deleted, as was explained in this section. The rest of the immediate transitions are simply discarded, since they are all immediate transitions without any triggers. This causes the delayed transitions originating in `d.a` and `e.a` to sequentially update their target from `d.b`/`e.b` to `d.c`/`e.c`, then to `f` and then to `g`.

Running example after eliminating transient states

Figure 3.19 shows the visualization of the running example after eliminating transient states. The transient states were eliminated. This includes the immediate transition connecting the (transient) case closing node to the node deciding the control flow according the if-else conditions of the Blech code. In terms of the stateful nature of the code and minding the necessary control flow elements, a minimal visualization of the given Blech code was achieved.

Chapter 4

Extending the compiler

The module for translating Blech code into textual SCCharts is integrated in the Blech compiler. The implementation of the translation tool can be found on Github[1].

4.1 General information

The translation tool is attached in the Blech compiler after the type checking step, i. e., the semantic analysis of the compiler. Hence, the translation tool has a typed abstract syntax as tree as input. It is given via the type `BlechModule` in the compiler. This type contains a list of all activities and functions of a Blech file, as well as a link to the activity marked as the entry point. The concept introduced before uses this list of activities. Functions given by this list will be ignored. Functions are not to be visualized as they do not have stateful behavior by nature. A single activity is given by the type `SubProgramDec1`. This type contains all relevant information needed for translating the Blech code. It contains information about the name, input and output variables, and other information, including its body, of course. The body is given by an ordered list of statements, which are represented by the type `Stmt`. Through this list, the body can be sequentially translated. Hierarchical elements contain a list of statements as their bodies as well. The type *Stmt* and its possible variations are given in Figure 4.1. As described in Chapter 3, the cases `await`, `ITE`, `cobegin`, `while-repeat`, `repeat-until`, `preempt` and `ActivityCall` are relevant. These are the constructs apart from activities that have defined translation steps in the translation. When translating an activity, its body can be translated sequentially and recursively with the defined translation rules. We can see that the hierarchical elements contain bodies: lists of the type `Stmt`.

The general overall workflow in the translation tool is as follows:

1. Parse the given Blech types and translate them into an abstract graph structure. One activity is translated into a graph.

2. Execute the defined simplification steps on the abstract graph structure.

3. Translate the graph structure into a textual SCChart.

4. Output the sctx-file into the present working directory.

[1] `https://github.com/dalu2104/blech`

```
type Stmt =
    | VarDecl of VarDecl
    | ExternalVarDecl of ExternalVarDecl
    | Assign of range * TypedLhs * TypedRhs
    | Assert of range * TypedRhs * string
    | Assume of range * TypedRhs * string
    | Print of range * string * (TypedRhs list)
    | StatementPragma of Attribute.StatementPragma
    | Await of range * TypedRhs
    | ITE of range * TypedRhs * Stmt list * Stmt list
    | Cobegin of range * (Strength * Stmt list) list
    | WhileRepeat of range * TypedRhs * Stmt list
    | RepeatUntil of range * Stmt list * TypedRhs * bool
    | Preempt of range * Preemption * TypedRhs * Moment * Stmt list
    | StmtSequence of Stmt list
    | ActivityCall of range * QName * Receiver option * TypedRhs list *
        TypedLhs list
    | FunctionCall of range * QName * TypedRhs list * TypedLhs list
    | Return of range * TypedRhs option
```

Figure 4.1. The type Stmt in the Blech compiler. It defines all possible statements in Blech.

In order to be able to make simplifications on the graph, the given Blech code is translated into an abstract graph structure conveying the needed information to generate textual SCCharts afterwards. Since SCCharts are specified textually, it would be hard to parse and output sctx-files after every simplification operation. Hence, the helping graph structure is used. In later implementations, the graph structure can also be used to add a different visualization instead of SCCharts. This thesis focuses on SCCharts only, however.

In the translation module, the four phases are realized in separate files: Translation.fs, Simplification.fs, SctxGenerator.fs and SctxToFile.fs. The file BlechVisGraph.fs offers data types as well as some helpful methods. Visualization.fs is the main method of the module and orchestrates the phases and takes care of the configuration options.

4.2 Abstract Graph Structure

To model the abstract graph structure, an already defined library in the compiler was used: *Blech.Common.GenericGraph*, which is, as the name conveys, a library for a generic graph. It offers the possibility to define a graph with nodes and edges and their specific payloads, as well as some common graph operations (join, copy) on top of the basic operations to create a graph (add, remove, get successors, etc.).

States are nodes and edges are transitions in the abstract graph. Their respective payloads

convey the informations needed to generate the wanted visual representation of the code. The payloads are very complex and detailed. Hence, only a small example of what is conveyed in the payloads is given. The details of the abstract graph structure can be found in the file *BlechVisGraph.vs* in the translation module. The payload of a node, for example, contains a field that indicates their initial or final status. Complex states contain a node that indicates their complexity. If they are a complex node, another graph is contained that represents the inner-behavior of a hierarchical element, such as an if-branch in Blech. The inner-graph will results in an inner-behavior of a complex state in the final SCChart. An edge payload, for example, contains a field for the label of the edge and its type (abort, immediate, delayed, etc.).

The nodes, edges and specific payload properties are set and added to the graph while sequentially parsing the single statements of the given Blech code as described earlier. Label specifying statements are parsed here as well, and carried along until they are assigned their correct state according to the rules in the concept.

The generated abstract graph is then simplified with the defined simplification steps on the graph. For example, flattening the hierarchy of if-else constructs is done by elevating the contained graph of states representing if-elses to the higher abstraction level and connecting incoming and outgoing transitions to the former inner initial state and final state, respectively. The simplification steps can simply be executed, because the graph was designed with the resulting SCCharts in mind.

After the simplifications, each graph is parsed starting at the initial node. The nodes and edges are translated into their textual SCChart elements according to their specified properties. The graph is parsed by visiting the connected nodes via edges. Textual SCCharts are specified the same way: a state followed by its outgoing transitions. Complex nodes can be translated into textual SCCharts hierarchically, as SCCharts are hierarchical as well.

4.3 Challenges

When parsing the Blech code into the abstract graphical representations, the conditions and activity call parameters are just parsed as labels for edges, if they are composed boolean conditions, i. e., `variableA and variableB`. In SCCharts, this needs to be changed to `variableA && variableB`. Hence, during the generation of textual SCCharts, the composed boolean conditions are parsed and splitted, and the Blech specific keywords are replaced by SCCharts specific ones.

Another issue regarding specific keywords occurs when activity names or variables contain SCChart keywords. This will cause issues in the SCCharts-compatible editors when importing the textual SCChart representation. Hence, variable and activity names are parsed for SCChart keywords. If such a keyword is present, an underscore is added beforehand.

The need for local variables is determined when parsing the Blech code. They are needed for run statements and boolean conditions referencing local variables in Blech. These variables need to be realized as variables in the SCChart as well, or else there will be errors. So, when

a statement referencing such local variables is detected while parsing Blech code, the local variables with name and type are aggregated. When the body of interest is parsed completely, the aggregated local variables are added as a field to the node containing the examined body. This way, the local variables can be added at the beginning of a state in the textual SCChart, which is needed in the SCCharts syntax. The variables, which are then used to reference SCCharts or evaluate boolean conditions, are present and no errors are presented by the SCCharts compiler.

Another issue that is present when generating SCCharts is that the identifiers for states in the textual SCChart need to be unique. Hence, a running counter is incremented while parsing the Blech code and generating the abstract graph. Each state in an activity is then assigned a unique number as an identifier. This causes no issues until it is desired to inline activities due to flattened run statements. One could not restart the running counter for node identifiers for every activity, since activities can be called by multiple run statements. So, just copying the activity twice will cause multiple nodes with the same identifier to be present. Hence, a second identifier is added as a property for the states. This second identifier is set during the simplification step of flattening hierarchies. The hierarchies are flattened from the inside, so from the innermost statement, to the outside. The secondary identifier can only be set exactly once, so nested run statements do not override previously set secondary identifier. Thus, having run statements to be nested and calling an activity with multiple run statements will no longer cause issues with this introduced secondary identifier.

Lastly, it is to be mentioned that some feature have not been implemented due to time-constraints. Not implemented are the alternative weak-abort pattern, reducing the complex node with multiple regions by the awaiting region for more than two regions, and the extended labeling feature. The alternative weak-abort pattern is realized for exactly two regions only, and labeling is limited to labeling states.

4.4 Configuration

The configuration options for the translation tool are listed in Table 4.1. With no options set, the default options will be used. The default options were chosen according to the evaluation that will be explained in a later chapter.

Table 4.1. Configuration options of the translation tool.

Flags	Description
`--vis-breakrunstatement` `-vis_breakRunStmt`	With this flag activated, the activities referenced in run statements will be inlined in the visualization. Recommended for small programs and activities only.
`--vis-cbgnpatternwithhier` `-vis_cbgnPatternWithHier`	Uses an alternative visualization for a recognized pattern in cobegins (modeling a weak abort). This will have a hierarchical visualization of said pattern, in comparison with the hierarchy breaking alternative.
`--vis-cbgnpattern` `-vis_CbgnPattern`	This flag will cause the cobegin pattern described above be recognized but with a hierarchy-less visualization.
`--vis-includeorigcode` `-vis_includeOrigcode`	With this flag set, the original Blech code will be included as a comment in the visualization.
`--vis-notuseconnector` `-vis_notUseConnector`	With this flag set, connector states will not be used in the visualization instead of simple states whenever possible. Not recommended to use.
`--vis-disablebreakhier` `-vis_disableBreakHier`	Disables the simplification step of breaking introduced hierarchies. Not recommended to use.
`--vis-disablecollapsetrans` `-vis_disableCollapseTrans`	Disables the simplification step of breaking introduced hierarchies. Not recommended to use.

Chapter 5

Evaluation

This chapter evaluates the concept and results of the implementation described in Chapter 4. Since the goal of this thesis is to achieve an extraction and illustration of the implicitly given states of the Blech code, quantitative metrics were not used to evaluate the result. There is no quantitative goal that was to be reached, or is meaningful enough to be measured laboriously. The concept and implementation is evaluated by the assessment of small synthesized programs that show debatable scenarios and different options, as well as exemplary Blech programs that can be rated regarding states in the code and usefulness of the visualizations. In addition to my own assessment, a group of experts on Blech and Statecharts was consulted and their assessment of the described visualized Blech programs is evaluated.

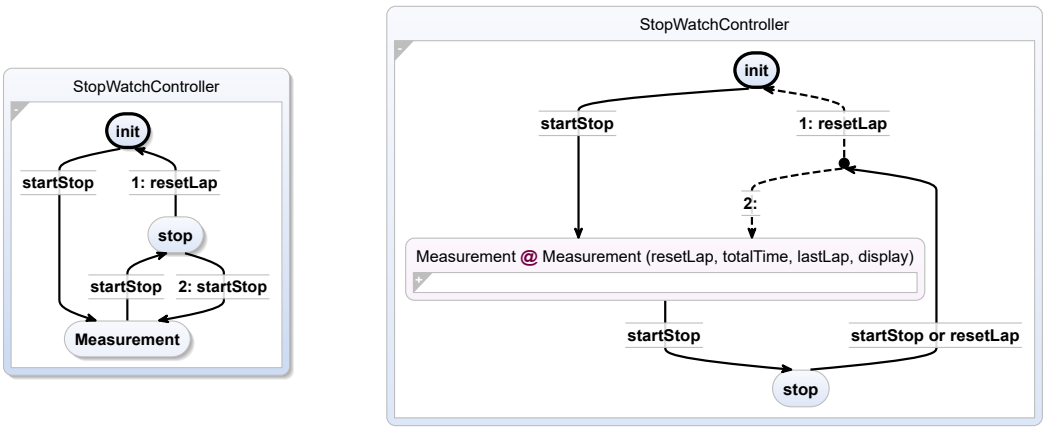
5.1 Empirical evaluation

This section looks at different scenarios as well as real Blech code examples and identifies troubles and possible improvements. The Blech files and generated .sctx are available on Github[1]. The real Blech programs can be found in the programs sub-directory and are the following: a controller for a blinker of a car (*blinker.blc*), a controller for a water pump (*control.blc*), an implementation of the game react (*react.blc*), an implementation of a stopwatch (*stopwatchLAPFAST.blc*) and a virtual safe lock (*virtualSafeLock.blc*). Images of the models used in the evaluation are also included in the appendix. Note, however, that SCCharts offer the ability to expand or collapse complex states. Therefore, some parts of the images are hidden. To be able to view all the parts of the illustration, the usage of a compatible editor is needed.

Figure 5.1a shows the envisioned intuitive result that was used in the thesis proposal. It is what was envisioned to be the result of the given Blech code. Figure 5.1b shows the illustration of the same Blech code, but with the implementation of the concept given in this thesis. The shared source code is given in Figure 1.1 in the introduction. The most noticeable differences are the presence of hierarchy in the node with the label *Measurement* and the possibility to view the diagram inside. Further, an additional connector node is present that is the origin of the splitting control path. The envisioned illustration merged the previous awaiting edge and the conditional edges and logically assigned the conditions. Overall, the result is a compact drawing that visualizes the existing states in the code.

Another interesting case is given by an implementation of *react*. *React* is a game, including menu guidance with a help, result and high score screen. Thus, the different activities are

[1] https://github.com/dalu2104/blech-mode-extraction-evaluation



(a) The illustration as given in the project proposal.

(b) Resulting illustration of the code of the project proposal by this thesis' concept.

Figure 5.1. Illustration given in the project proposal and the illustration of the same Blech code resulting by this thesis' concept.

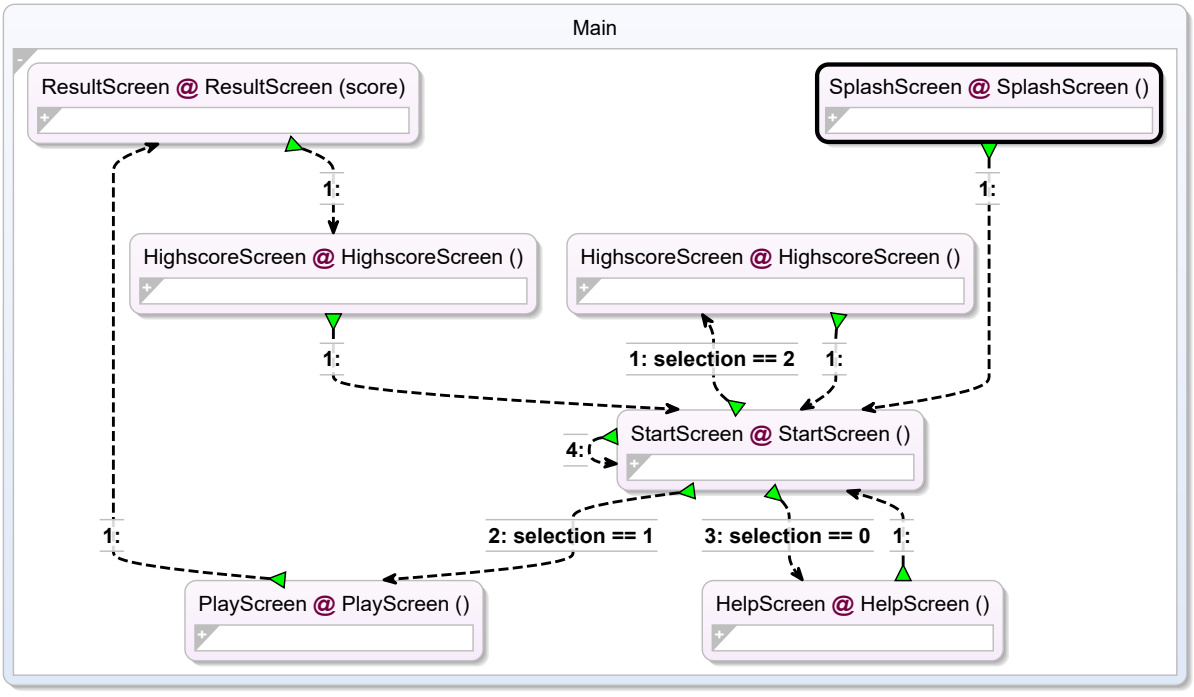


Figure 5.2. Visualization of implementation of the game called *react*. This version does not have flattened run statements (referenced SCCharts). The flattened version can be found at Figure 5.3.

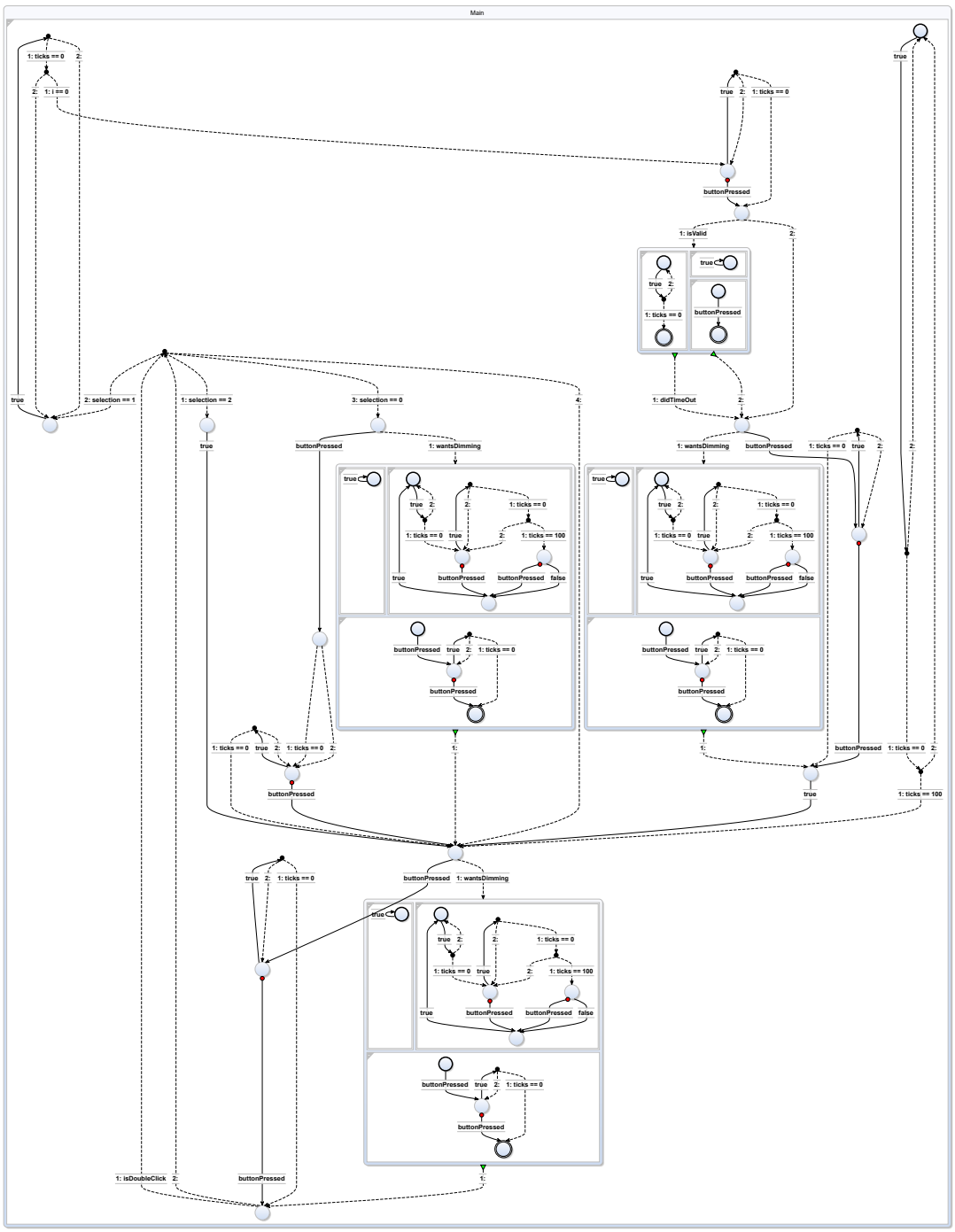


Figure 5.3. Visualization of implementation of the game called *react*. This version has flattened run statements (referenced SCChart). The unflattened version can be found at Figure 5.2.

more or less a chained sequence of run statements in the Blech code and referenced SCCharts in the processed visualization. As can be seen in Figure 5.2, maintaining the given hierarchy of activity calls causes the graph to be more tidied up. Certain activities can be expanded on demand, when they are of interest. Flattening run statement hierarchy by default can cause important separation of concern to disappear, resulting in complex graphs, where information about states is not easily detectable and link-able to their context. The flattened version of the visualization can be found in Figure 5.3. Due to the nature of the control flow, which is managed through user input, the flattened version has many conditional immediate transitions, detaching the result from the intuitively best one. Further, the flattened version is more chaotic and there is no context given for the different parts of the diagram. For this example, flattening by default is not desired.

Regarding the weak-abort pattern present in cobegin constructs in Blech as described in Section 3.3.1, it can be observed that it works well for the real Blech programs evaluated. The pattern was matched in multiple instances, for example in the activity *EnterNewSecret* in the virtual safe lock example. This is illustrated in Figure 5.4. Figure 5.4a shows the Blech code containing an activity with a cobegin construct, where one branch is weak and the other only contains an await statement. Figure 5.4b shows the corresponding visualization, where the pattern was matched and the hierarchy of the cobegin was flattened. In comparison, Figure 5.6 shows the visualized code. However, in this version, the pattern was not matched, and therefore no flattening was done. The flattened version is more compact as it has less hierarchy and its representation of the stateful nature of the code is better visualized. Moreover, the user does not need to understand the semantics behind concurrent and final regions in SCCharts.

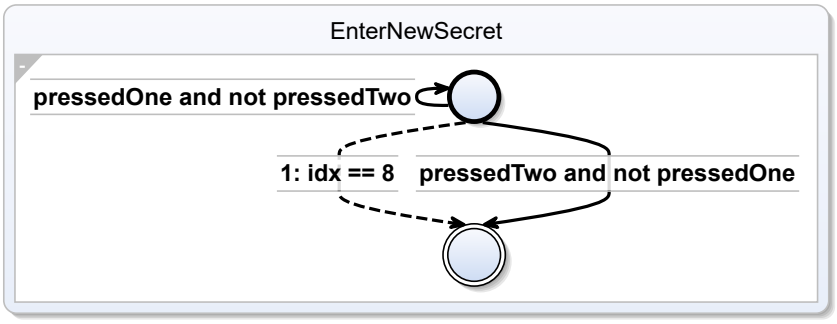
In Section 3.3.1, it was planned to extend the weak abort pattern to multiple non-awaiting branches. In the Blech program examples, there are instances where this concept could be applied. One of them is illustrated in Figure 5.5. A branch containing only an await statement is present, whereas the other branches are weak branches. As discussed in Section 3.3.1, however, one would need to identify that at least one branch is terminating in order to recognize the need for a termination transition. Further, the information about weak branches needs to be maintained, meaning that the user needs to recognize the meanings of final regions. This, however, is also the case if the pattern is not matched and the await region is not extracted as a weak-aborting transition.

Another noteworthy situation can be identified in the scenario presented in Figure 5.7. A transition is guarded by an expression that evaluates a comparison of a constant with a numerical value. This is not a dynamic calculation as the values are fixed and hence, it is evaluated in a compilation step previous to the visualization step. Ultimately, the evaluated expression, a boolean value of false in this case, is given as a guard for the transition and is presented in the visualization.

Lastly, there are instances to be observed where a separated control flow joins into a final state. An example is given by Figure 5.8. It was stated in Section 3.3.2 that a final node with multiple incoming transitions should not be deleted and reassigned. This case is observed

```
activity EnterNewSecret
(pose: nat32, pressedOne: bool, pressedTwo: bool)
(newSecret: [MAXLEN]nat32) returns bool
    var idx: nat32 = 0
    cobegin weak
        repeat
            await pressedOne and not pressedTwo
            if poseIsExact(pose) then
                newSecret[idx] = pose
                idx = idx + 1
            end
            // else inexact position, do not evaluate
        until idx == MAXLEN end
    with weak
        await pressedTwo and not pressedOne // finish
            programming
    end
    return idx > 0 // at least one position has been
        entered
end
```

(a) Code of the activity containing a cobegin matching the weak-abort pattern.



(b) Resulting illustration of the code containing a cobegin matching the weak-abort pattern.

Figure 5.4. A detected instance of the weak-abort pattern realized through a cobegin construct with an awaiting branch and a weak branch in the file *virtualSafeLock.sctx*. The illustrated activity is *EnterNewSecret*.

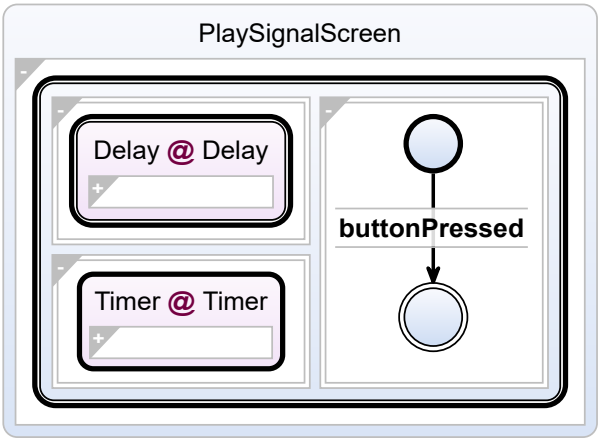


Figure 5.5. Visualization of activity *PressToContinueController* in the game called *react*. The activity contains a cobegin with an extended weak-abort pattern that could be simplified further.

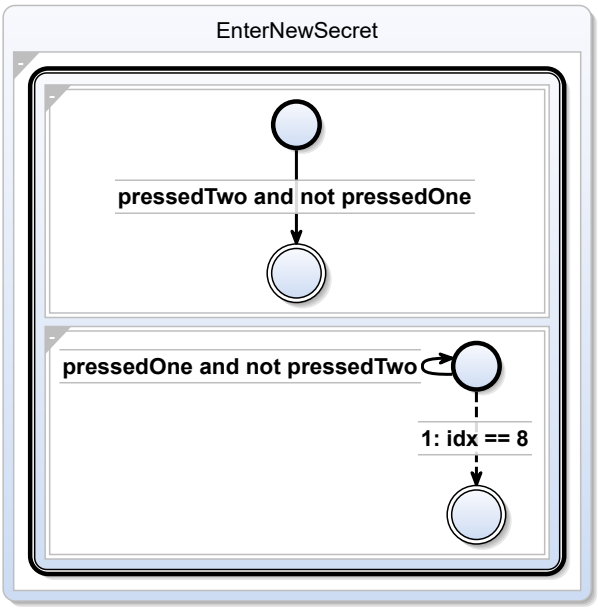
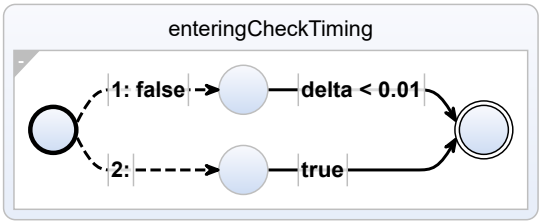


Figure 5.6. A non-flattened weak-abort pattern realized through a cobegin construct with an awaiting branch and a weak branch in the file *virtualSafeLock.sctx*. The illustrated method is EnterNewSecret. The corresponding code is given in Figure 5.4a.

```
const C_VehicleState: float32 = 0.0
@[EntryPoint]
activity enteringCheckTiming
(delta: float32)
    if (C_VehicleState >= 2.0) then
        await (delta < 0.01)
    else
        await true
    end
end
```

(a) Code of an activity that checks a constant against a fixed value in a conditional control flow.



(b) Visualization showing the evaluated expression checking a constant against a fixed value.

Figure 5.7. An example program and corresponding visualization showing how an expression containing only constants and numerical values is evaluated in earlier compiler steps, resulting in the visualization showing the evaluated expression.

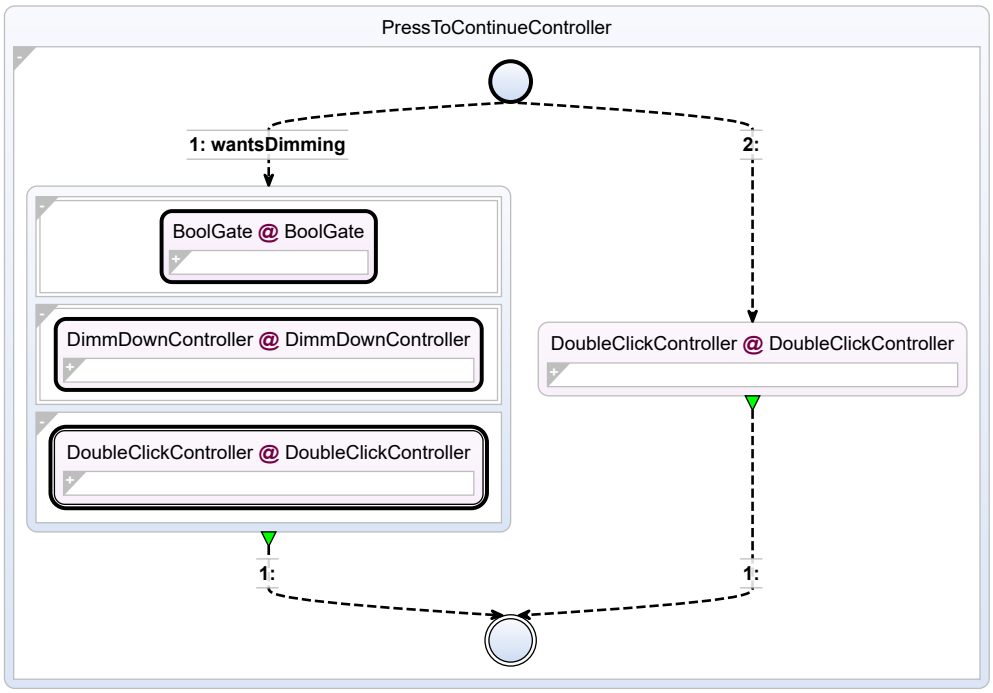


Figure 5.8. Visualization of activity *PressToContinueController* in the implementation of the game *react*. The control flow merges into a single final state. Further simplification in terms of reducing the amount of states and transitions is possible by removing the final state and its incoming transitions and reassign the final states to the previous two states.

and is a possible target of further simplifications. This simplification would save further unnecessary nodes but increase the amount of final states. This is a consideration that has to be made when introducing this simplification.

5.2 Expert survey

This section covers the expert survey aiming to evaluate this thesis' concept. The artificial and real-world Blech programs are identical to the ones used in Section 5.1 and are available on Github[2] or in the appendix. The Github repository also contains a document for instructions on how to use the visualization module implemented in Chapter 4, how to read state charts and SCCharts, a link to install a SCCharts compatible editor (KIELER or KEITH) and most importantly an explanation on the contained Blech programs and their visualization and corresponding questions that are to be answered. Remember that the images that are available in the appendix are partly collapsed and should rather be explored in a compatible editor.

The interviewed group of experts consists of four experts in Blech programming and in the field of synchronous programming languages. They are partly in research positions at the University of Kiel and the corporate research department of the Robert Bosch GmbH. Three of the experts are in-depth experts on the Blech programming language.

5.2.1 Survey Details

The survey is divided into six parts, each covering a specific topic. The six topics are: proposal example, real Blech programs, abort construct hierarchy flattening, run statement hierarchy flattening, weak-abort pattern in cobegin constructs and labeling alternatives. The single topics are explained in detail below. Every topic allowed for free text answers, giving the experts the opportunity to freely voice their opinions and concerns, regardless of scenario specific questions.

Proposal Example

This thesis was inspired by a draft of a state chart that was created by hand illustrating the intuitive states contained in a Blech program. With the desired result in mind, the concept of this thesis was developed. The desired result is given in Figure 5.1a and the corresponding Blech code was given in the introduction in Figure 1.1. In this part, the experts were asked to assess the generated result and compare it to the desired result. The goal here is to determine whether this thesis produces comprehensible results that satisfy the expectations, which version of the two is better liked and therefore what conclusions can be drawn about what differences are identified and are desired to be implemented.

[2] https://github.com/dalu2104/blech-mode-extraction-evaluation

Real-world Blech Examples

The second part covers other real Blech code examples that are, in comparison with the following examples, not constructed by hand to show specific situations. The real code examples were taken from the official Blech language website[3]. In this part, there are no comparisons to be made as there is no predefined result in mind, nor is there a debatable alternative version at hand. Constructing an alternative version by hand proves to be a very hard task, considering the complexity of the programs. This part was probably the most difficult for the experts, as there is no set boundaries or specific questions to be asked. The goal is a general assessment of the quality of the drawings and identification of problems, troubles or strong points. The experts were asked to give a free text answer, allowing them to freely express their concerns and thoughts. They were also asked, if in their opinion the visualizations are viable representations of their corresponding code and whether the visualization serves its purpose to understand the nature of the given Blech code.

Flattening of Abort Statements

Part three covers the flattening of abort constructs. They experts were given two visualizations of the same simple Blech program. The program contains an abort construct with simple inner behavior. The two alternatives are an unflattened and flattened visualization of the abort construct, respectively. The general alternatives were described in Section 3.3.1. The experts were asked to determine whether the alternatives are viable representations of the given Blech code. They were also asked to name the version they like better and if their opinion would change if they imagined a more complex inner behavior in the abort construct. Lastly, a free text question was asked to give the opportunity to express opinions and concerns freely.

Flattening of Run Statements

The fourth part covers the flattening of run statements. As discussed in Section 3.3.1, run statements are generally visualized hierarchically with the option to flatten the hierarchy, elevating the inner graph to the higher hierarchy level. This part gave an example of a simple Blech program containing a run statement. The experts were given two version: one flattened and one unflattened. The experts were asked if they found the two options viable representations of the Blech code, and to assess which one they preferred and why. As previously, they were given the opportunity to freely express their opinions and concerns. They were also asked whether or not a more complicated inner behavior of the referenced activity would change their opinion about what version they like better.

Weak-Abort Pattern

Part five covers the weak-abort pattern as discussed in Section 3.3.1. The given Blech program contained a cobegin construct that matches the described cobegin pattern. The experts were

[3] https://www.blech-lang.org/docs/examples/

given three alternative visual representations: (1) a direct unflattened version, showing all branches of the cobegin regularly, (2) a flattened version, where the awaiting branch has been removed, the weak-branch elevated and the await-transition been added as an aborting transition to the inner behavior, and (3) the alternative representation, keeping the hierarchy, removing the awaiting branch and adding said transition as a weak-abort transition to the hierarchical node. The experts were asked if they found the three alternatives to be viable representations of the given Blech code. They were also asked to assess which version they liked best and why and were also given the chance to freely express their opinions.

Labeling

The last part covers with labeling alternatives as discussed in Section 3.2. One of the the two given versions shows the alternative focusing on labeling real states by collecting and merging the labels in their respective state. The other version allows more flexible labeling, such as labeling cobegin hierarchical states and regions in cobegins by adding further keywords and only allowing one state label per state. As before, the experts were asked to assess the viability and what option they prefer and why. Additionally to the free text answer, they were asked to state if they would make use of the extended labeling feature if it were given to them.

5.2.2 Survey Results

In this section, the results yielded by the expert survey are analyzed. The main points of these findings will be discussed in detail in Section 6.2.

General feedback includes that the visualization is generally very helpful to understand the stateful code and its structures. Structures in the code are easily detectable most of the time and reused elements are easily detectable according to the experts, such as multiply called activities. Also, the general feedback and conclusion of statements made by the experts suggests that hierarchical illustrations (abort, run, cobegin) are generally preferred. In some cases, however, it is useful to flatten them. Therefore, the experts confirmed the benefits of being able to configure the visualization.

Multiple experts, especially those without prior SCCharts experience, stated SCCharts knowledge is needed to really evaluate the results and understand what is going on in the visualizations precisely. This is a rather important observation, as consequentially, the visualization tool cannot simply be used "out of the box". Even a future Visual Studio Code integration for the purpose of programming and documentation support is influenced by this observation. A detailed discussion about this point is to be found in Section 6.2.

Another general remark was made in terms of relating elements of the code to elements in the visualization and vice versa. It was noted that, especially for complicated code bases, it would be helpful to be able to click into the visualization and be guided to the exact piece of code and vice versa in the visualizations. Of course, such feature can only be considered for future IDE integration, i. e., Visual Studio Code, not requiring a whole tool change as it is now.

Furthermore, the feedback revealed that there were issues to install the SCCharts editors KIELER and KEITH on the operating system MacOS. Hence, two of the experts were not able to explore the visualizations in the tool, but had to resort to using images of the visualizations for the evaluation. The images showed all activities and did not offer the option to expand referenced SCCharts.

Proposal Example

All experts confirmed the validity of the generated visualization and its helpfulness to understand the stateful nature of the code.

The main difference between the two presented options was the joined transitions of an await statement followed by a decision point resulting of a repeat-until statement. Most experts actually preferred the merged transitions version. Their argument considered the need to evaluate the conditions of the control point and await statement in their head to know under what conditions the transitions are taken. With the joined version the path is immediately clear. The expert that preferred the non-joined generated version argued that the non-joined version is closer to the code, representing the statements of the code and thus, the hurdle to identify the visualization from the code is smaller. Consequentially, most experts argued for an implementation of the joined-transitions. The free text question yielded that the intuitive approach can be considered more of a sketch to the program, whereas the generated version actually documents the given code. An expert criticized that the types for local variables (needed for referenced SCCharts) are not yet given. Further, the same expert noted that a program containing only an await statement in a loop does not convey any information when visualized. The expert proposed to somehow indicate whether the await statement operates before or after some expressions, so the user will be able to determine the context and purpose of the await statement.

Real Blech Examples

All experts confirmed the validity of the generated visualization and its helpfulness to understand the stateful nature of the code.

While examining the visualizations of real Blech code examples, multiple experts, especially those without (detailed) SCCharts knowledge, stated that experience in SCCharts is needed to understand the visualizations completely. States, transitions and hierarchy are easy enough, but in terms of the difference in transitions (abort, terminate, delayed, immediate, etc.), there was some uncertainty to be found. This implies an issue, since the tool might not be suitable to be used "out of the box". There has to be some form of learning for the user before the tool is used.

Experts also expressed their dislike of how parameters are represented in SCCharts, as there is no visual difference between in- and output variables as it is in Blech. As mentioned in earlier chapters, this distinction is an important part of the Blech programming language.

Next, an expressed concern regarded the labels in referenced SCCharts caused by run statements in Blech. Currently, the name of the called activity is added as a label to the state. Since the state is a referenced SCChart, the name of the SCChart that is referenced is added via *@NameOfChart* behind the state's label. Consequentially, in a state caused by a run statement, the label will show *NameOfChart@NameOfChart*, which looks clunky. However, if the complex state is merged with a following labeled await statement and the name of the activity was not explicitly labeled, only the label of the await statement would show on the complex state. That would not be an intuitive labeling as it would primarily hint at the awaiting state.

Regarding highlighting special elements, statements were made that it would be possible to specially mark external variables. External variables in Blech are external variables given to the Blech program via the environment. Currently, regular and external variables all look the same in the visualization. Some extra hint for external variables would be beneficial. Another extended visual clue could be made for singleton activities, an expert stated.

Further, the identified and discussed extended merge of transitions, as was discussed in the paragraph before, would have uses in the real Blech code examples, stated the experts. This further accumulates the arguments to implement this feature in the future.

An issue that was identified are the compiler pre-evaluated elements of the Blech code, such as the expression involving a constant mentioned in Section 5.1. Another pre-evaluated element is the when-reset construct. It is transformed into a repeat until inside a when-abort in previous compiler steps. Such pre-evaluation is out of the scope of this thesis and nested inside previous compiler steps. Unexpected transformations naturally estrange the visualization from the code and might take the user by surprise, which is supported by the expert's dislike for the pre-evaluated elements. In the future, it would be beneficial to find a way for the compiler to not execute certain pre-evaluations.

Further, an expert expressed surprise about the realized weak-abort pattern in the code examples, as the real Blech code examples were generated with the regular weak-abort pattern flattening enabled. This might have happened due to the fact that said developer did not know about the pattern before as the pattern is specifically handled in a later section of the evaluation. Nonetheless, it is an important point to be made as the flattened version of the pattern does not precisely represent the given code but rather makes a simplification that is not immediately detectable when the user is inexperienced with the tool and especially the realized weak-abort pattern. This is a strong argument for not having the flattened weak-abort pattern as the default setting.

Lastly, it is to be mentioned that this part unveiled a bug, where certain states were not recursively checked in terms of the collapse of immediate transitions. This bug was promptly fixed.

Flattening of Abort Statements

All but one expert confirmed the validity of the options. One expert did not approve of the flattened version and argued that the hierarchy is very much needed for abort statements and it is also closer to the the given Blech code, which is why he preferred the hierarchical option.

Another expert stated that the flattened version is only viable for rare specific cases and small inner behaviors. Therefore, this supports the option to be able to configure the flattening of abort statements.

Flattening of Run Statements

The experts confirmed the validity of the options. Most experts clearly stated their preference for the hierarchical version and argued that they like to think locally. This was expected as the activities were extracted and separated in the code for reasons of separation of concern. One expert, however, can imagine specific use-cases where it would be beneficial to be able to use the flattened version. Therefore, the possibility to configure this feature was confirmed.

Weak-Abort Pattern

The experts again confirmed the validity of the options. In terms of preference, the expert made diverse statements. Each version had an expert that preferred it. The argument for the flattened version was the collapse of hierarchy. The alternative version was attractive due to the slimmed down hierarchy and extraction of abort condition. The argument for the regular version was its closeness to the code base, which is a strong argument. All things considered, the users should use their ability to configure the layout for their personal preference. Where the experts are united is their dislike for the indication of weak branches. They find the visual clue of final regions to be too subtle. This is an important point and should be considered in future work. The suggestions mentioned the use of different colors for the weak or strong regions.

Labeling

Generally, the advanced labeling feature was preferred by the experts. It was stated that the label declarations might lead to conflicts in bigger projects, which is problematic. Further, it was noted that the declared labels have to be maintained manually, which, in terms of software evolution, might be problematic because the labels need to be considered if the code base changes. If the code base changes, however, the visualization tool might not be used, but the declaration of the labels has to be minded. This is a likely source of errors and will lead to confusion, once the label declaration and code are not in sync anymore. A described example was a cobegin-construct that is given another branch. Not only has it to be remembered to update a label declaration but further issues such as the order of declarations has to be minded. Overall, this is a likely source of errors. It was further noted that the labeling is pretty tedious to use, when each element has a distinct label keyword to be used. One expert criticized the extended feature and would like a distinct label placement, i. e., labeling cobegin branches inside the branches, and not in front of the cobegin-construct. Lastly, multiple experts criticized the duplication of labels in the basic labeling version, pushing and duplicating a label into multiple cobegin branches. They prefer to discard the given label coming before the cobegin.

Chapter 6

Conclusion

This chapter gives a brief summary of the research question at hand, revisits the work and evaluation done, discusses the findings of this thesis, and lastly proposes work to be done in the future.

6.1 Summary

The research done in this thesis tries to realize an automatically generated visual representation abstractly showing the implicitly contained states and transitions of any given Blech code. The visual representation is meant as a support tool for Blech developers, giving them a visual clue about their code and serving as a documentation feature. For the visual representation of the code, SCCharts are chosen. SCCharts are a visual programming language, with all the tools for converting a textual to visual representation in place with editors such as KIELER[1] or KEITH[2]. Furthermore, both programming languages reside in the same domain of embedded real-time systems and are synchronous languages. The requirements for the programming languages are similar, so the choice of SCCharts was obvious, as its graphical elements are suited for the specific requirements given by the previously mentioned domain. To achieve a semantically equivalent translation is future work.

With this thesis, I propose an abstract translation from Blech to SCCharts. Statements in Blech are translated into SCCharts constructs recursively. The resulting SCChart is altered in two simplification steps in terms of hierarchy and unnecessary immediate transitions that are introduced during the synthesis. In some cases, alternatives of visual representation of certain Blech statements are presented. One such is given by an identified weak-abort pattern, where the Blech cobegin construct to model an implicit weak-abort. Three different options are presented in this case.

Next, the proposed concept for abstractly visualizing Blech code with SCCharts is implemented. The existing architectures given by Blech and the KIELER research projects are used. A tool is added to the Blech compiler chain, taking the compiled typed abstract syntax tree of the given Blech code and transforming it via the mentioned steps above into a textual SCChart (.sctx). In the realized workflow, the textual SCChart is to be imported into an SCChart-compatible editor, such as the options mentioned above. Then, in the editor, the visual representation of the Blech code can be viewed, examined and explored. The tool is enhanced

[1] https://rtsys.informatik.uni-kiel.de/confluence/display/KIELER/Quick+Start+Guide

[2] https://rtsys.informatik.uni-kiel.de/confluence/display/KIELER/KEITH

by configuration options.

A group of experienced Blech developers was asked to evaluate given visualizations of given Blech code. They were given visualizations of real Blech code examples, as well as artificial examples that illustrate the different layout options as described earlier. The experts were asked to validate the given visualizations, describe what they do not like and whether the visualization was helpful to understand the given code. For examples with different options, they were asked to chose one they liked the most and explain their choice. General findings are that the visualization is helpful to understand the stateful nature of the code, that generally hierarchical illustration for elements of hierarchical nature such as abort, run or cobegin statements are preferred and that an *out-of-the-box* use of the visualization is challenging, due to the need to know the semantics of SCCharts.

6.2 Discussion

The general feedback was that the visualizations are helpful, especially if the code base is unknown or things are discussed with people who are not familiar with the code base. The generated SCCharts of the implemented tool offer a good solution, but are still not the optimal solution. A couple of issues are left and the validation from a big group of users is still remaining. Single points of debate are discussed in the following paragraphs.

The first point to be discussed is the issue of pre-compiled elements of the Blech code. These include expressions with constants or when-reset constructs. The main issue here is that the Blech compiler pre-evaluates these statements, so at the point of latching the translation tool to the compiler, the orignal elements are non-existent and there is no way to reproduce them or determine them in any way. Changes to the compiler would have to be made, which is an unclear amount of effort currently. Another possible solution would be to try to attach the translation tool into the compiler chain at another point. Currently the translation tool is placed in such manner, that the typed abstract syntax tree is given as input. This way all statements are given sequentially and recursively, just as is expected for the concept. Choosing another point requires in-depth knowledge of the Blech compiler and estimating the required effort to alter data in such form that is can be used for the translation tool is difficult too.

As was found in the evaluation, it is generally preferred to keep the hierarchies induced by run statements and aborts. It was also found that the opportunity to be able to flatten these hierarchies for certain use cases is useful as well. Therefore, the proposed configuration is desired and will be kept around. Configuration is needed for other debatable options, too. Sure enough, offering multiple options increases the workload for maintenance and development, but the advantages are clear.

The identified possible merge of awaiting transitions followed by a decision point (i. e., if-else, repeat-until) was assessed to be useful and a desired feature. The merge is not trivial regarding the triggers on the transitions. By simply merging the triggers from the delayed transition with the branches of the if-else, the label of the await would simply be duplicated. It is doubtful that this is an optimal and compact solution. Further problems arise, if variables

are used in the if-else conditions and the await statement. Complicated conditions might emerge this way. Boolean optimizations and smart simplifications have to be made here to make the merge viable.

The next point to be discussed is labeling. There are multiple options to realize labels in terms of their declaration, place and handling (merging, discarding, etc.). The feedback to the proposed concept and suggestions by the experts revealed that labeling is a difficult topic, where experts have very individual preferences. Each expert expressed their own preferences and made their own suggestions. In conclusion, labeling is a complex issues, where different people have different preferences and consider very different variations to be better. A quantitative validation of different variations with a broad group of developers would help to determine the best solution for default settings.

Another discussable point regards immediate transition optimization into final states. This thesis proposes not to optimize multiple immediate transitions coming into a final state and thus, maintaining a single distinct final state for any inner behavior. This decision was made under assumptions and would need further validation and might just be a matter of preference again.

An important observation is that the indication of weak branches of cobegin-constructs is too subtle. This is an agreeable statement, as the added rectangle for final regions in SCCharts is a light grey color on a white background. Options to highlight a weak branch visually could be an added state with a label specifying the weakness or an added comment to the region. Both solutions seem to be rather confusing and are not a distinct visual clue. Alternative suggestions, such as using a stronger color to indicate final regions in SCCharts, would be a very good solution for this problem. However, the design and visual options of the SCCharts language are way out of scope of this thesis.

Some other expert suggestions regarding the extra highlighting of important Blech elements, such as external variables or singleton functions, are an interesting points too. However, the bounds of SCCharts are present on this issue as well. There are limited options to highlight these things, with unclear benefits to be realized. A concept to highlight them, maybe even new visual elements, as well corresponding validations are needed for this to be realized.

The most important finding is the necessity to have knowledge of SCCharts to understand the visualization, which was confirmed by multiple experts. Some form of learning or guide for users needs to be available for the tool to do its job without requiring the user to do research on their own. A useful idea is to implement a guide into the visualization (once it is integrated in an IDE) like a map legend, explaining the visual elements in short distinct manner. The issue of needing to understand SCCharts semantics raises the question of course, whether such details are needed or if a basic state chart with the most basic transitions suffice. It is generally a trade-off, of course, between using the most basic elements and using more sophisticated elements. The previous option makes for a basic, probably easy to understand visualization. However, it is not possible to express complicated semantics this way. If a single type of transition were to be used, for example, it would not be possible to express different sorts of transition from state to state, such as aborts, immediate or delayed transitions. The

latter option uses more complicated visual elements, such as different sorts of transitions. The concept does use abstraction to not visualize all of the code already, such as simple expressions. Hierarchies of constructs such as if-else or repeats are simplified. Different transitions between states are kept, because the stateful nature of the Blech code was the focus of this thesis. Hence, it was decided that it is beneficial to illustrate the different transitions. To help the user understand the visual elements, a legend for the visual representation would be an important next step. Other helpful instruments such as a tutorial could be possible as well.

6.3 Future Work

This section presents suggestions for future work done on the technical implementation and the research topic in general.

6.3.1 General

In future work, it is necessary to implement a test framework to make sure the results stay the same when changes to the code base are made. Due to the enhanced features that are recommended later, changes are to be expected.

Optimizations regarding the tool chain and configuration in the compiler context are to be validated and checked as well.

Next, the generated textual SCChart, which is the output of the translation tool, is not optimized to be used other than to visualize the code with a compatible editor. The textual SCChart is not formatted to be good looking and matching common programming conventions. Therefore, it would be hard to use as a programmer for alterations. If a user desired to make changes to the textual SCChart in order to change the visualization to match the user's needs, it would be hard to do, since the code is not formatted to be used by real people. A possible help to format the produced code is already in place in the compiler through the *Blech.Frontend.PrettyPrint* module, which takes care of formatting the C code that is produced when compiling Blech code.

Implementing the proposed labeling, which allows cobegin-caused complex states and their regions to be labeled, is another proposed future work. Currently, only the basic labeling allowing awaiting states to be labeled is implemented. Implementing the extended feature is found to be beneficial in the evaluation.

Furthermore, the evaluation also finds that implementing the discussed merge of delayed transitions with conditional immediate transitions is beneficial. Decisions on the concept of the merge have to be made as merging the conditions leaves some questions as discussed earlier.

The next implementation detail that was found to be desired to be improved regards pre-evaluated statements in the Blech compiler. Some of these are unwanted by some users, for example when evaluating expressions with constants or the when-reset construct. They

prefer their visualization to be closer to the code base. Of course, being able to configure the amount of abstraction and pre-evaluation done by the compiler evaluations would be optimal.

6.3.2 Extending the Concept

For further work, this thesis proposes to develop a concept regarding the highlighting of external variables. As it is a rather crucial information for Blech programs, it would be helpful to highlight such variables in the visualization.

Another important point that the evaluation shows is that the highlighting of weak branches of cobegin-constructs as final regions in SCCharts is too subtle. Proposing a concept for different highlighting, such as giving the whole region a different color, should then be followed by implementing the proposed concept.

Further validations from a broad group of developers is needed to further validate the results of this thesis. The validation done here was realized with only a small group of four experts. Having a broader group of developers assess the findings would results in a stronger evaluation.

To take the next step, it would be possible to enhance the work done to a full semantically equivalent translation. The current translations have to be confirmed and missing Blech pieces such as simple expressions have to be added. The expressions can be added as an action to transitions or complex states. Current translation rules have to be checked since they did not have the semantic equality in mind.

6.3.3 IDE Integration

Chapter 1 introduces the vision of using the visualization while developing Blech code. Since the Blech tools are linked to Visual Studio Code, the idea is to have the visualization included in Visual Studio Code. This way, the user does not have to use multiple tools to access the visualization. The compiler of Blech has the ability to produce textual SCCharts. The future work needs to find a way to include the visualization tools of SCCharts in Visual Studio Code. Then, the textual SCChart resulting from the Blech compiler via the Blech tools can be visualized by the SCCharts visualization tools in Visual Studio Code. In this context, it is needed to determine the default KIELER layout settings. Also, a configuration panel for configuring the visualization is needed. If different layers of abstraction are identified, which handle situations closer to the code or more abstract and vice versa, they could be configured with a slider. This way, the user would be able to determine the preferred the abstraction level from very basic to the semantic equivalent version.

In such a Visual Studio Code integration, the possibility should also be given to link the elements in the code with their visual representation. Clicking in the visualization should highlight the corresponding parts in the code and vice versa. For example, this is already the case in KIELER when working with textual SCCharts and their visualization. An idea to connect

the visual elements to the code without having a IDE integration would be to note that line numbers of the code to specific visual elements.

The related work of this thesis showed a work done on translating the language Esterel to SyncCharts while maintaining the ability to view the translation steps as a user. This way, transformations can be understood and comprehended. Such feature of being able to view the abstract translation of this thesis step by step is a nice idea to ensure that the user comprehends the visualization and its relation to the code. This could also be part of the IDE integration.

6.3.4 Going beyond SCCharts

Lastly, it would be possible to use other forms of visualization for abstractly showing Blech code. The tool synthesizes an abstract graph structure to produce the textual SCCharts. This way any visual representation can be generated from this abstract graph. Using different state chart dialects or even proposing new Blech specific visualization elements would be options to consider.

Appendix A

Appendix

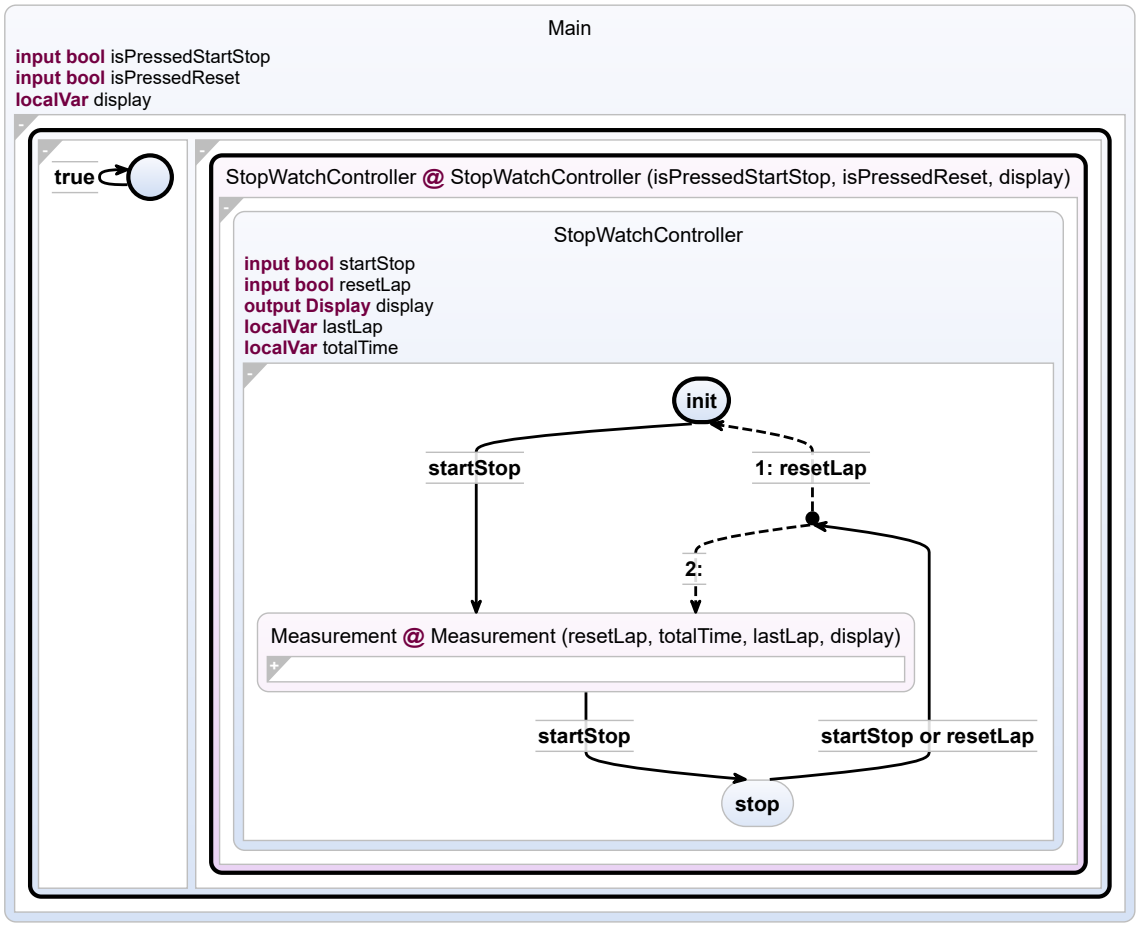


Figure A.1. Visualization of the project proposal code given by Figure 1.1.

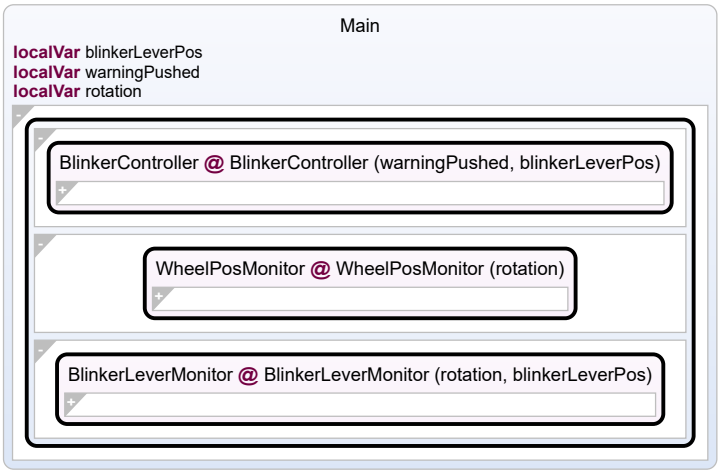


Figure A.2. Visualization of a blinker controller for the expert survey.

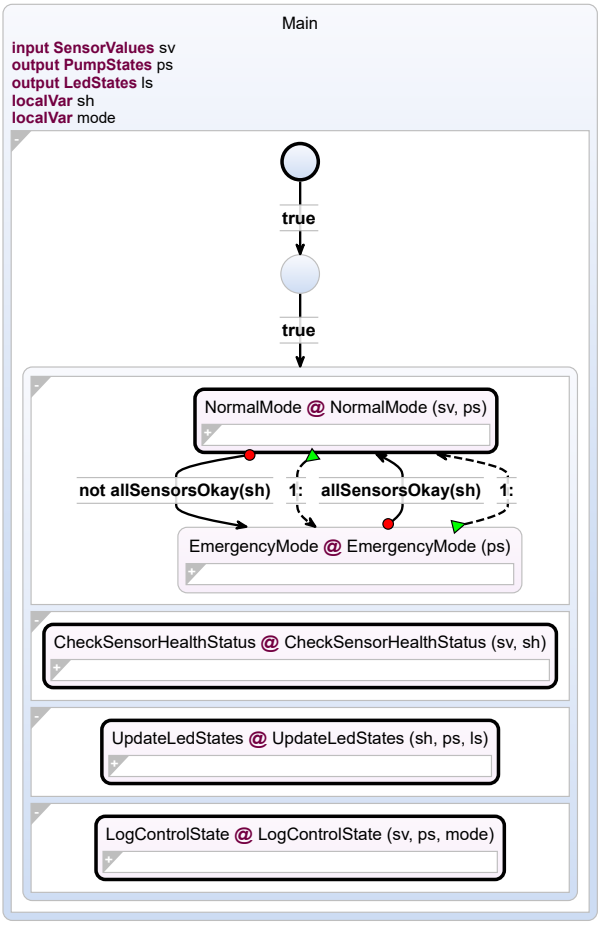


Figure A.3. Visualization of a pump controller for the expert survey.

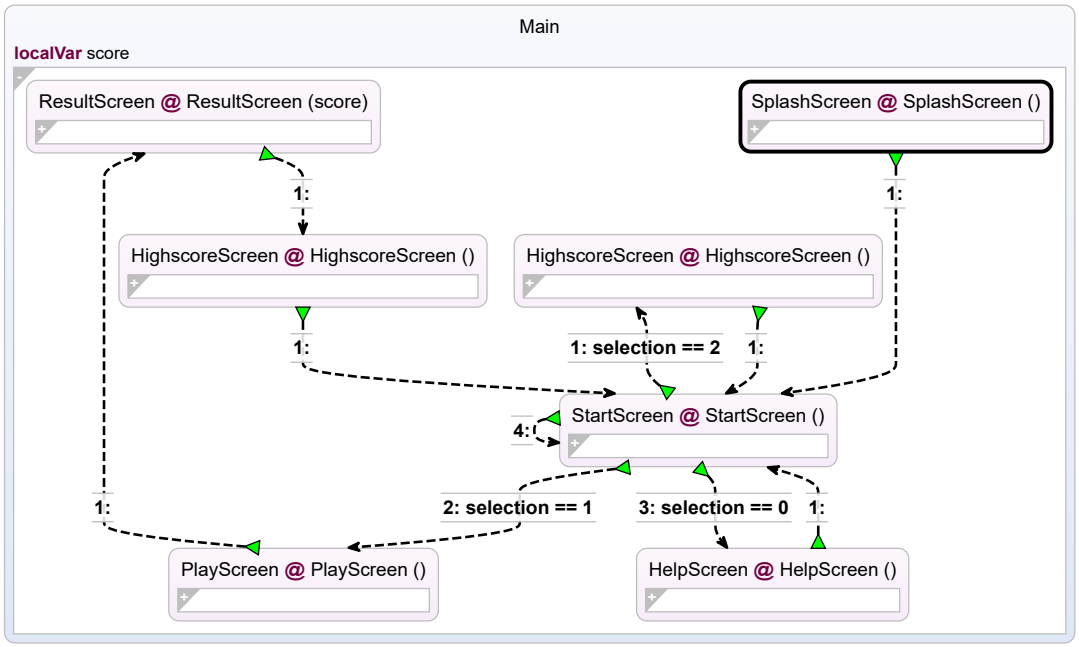


Figure A.4. Visualization of the game called *react* for the expert survey.

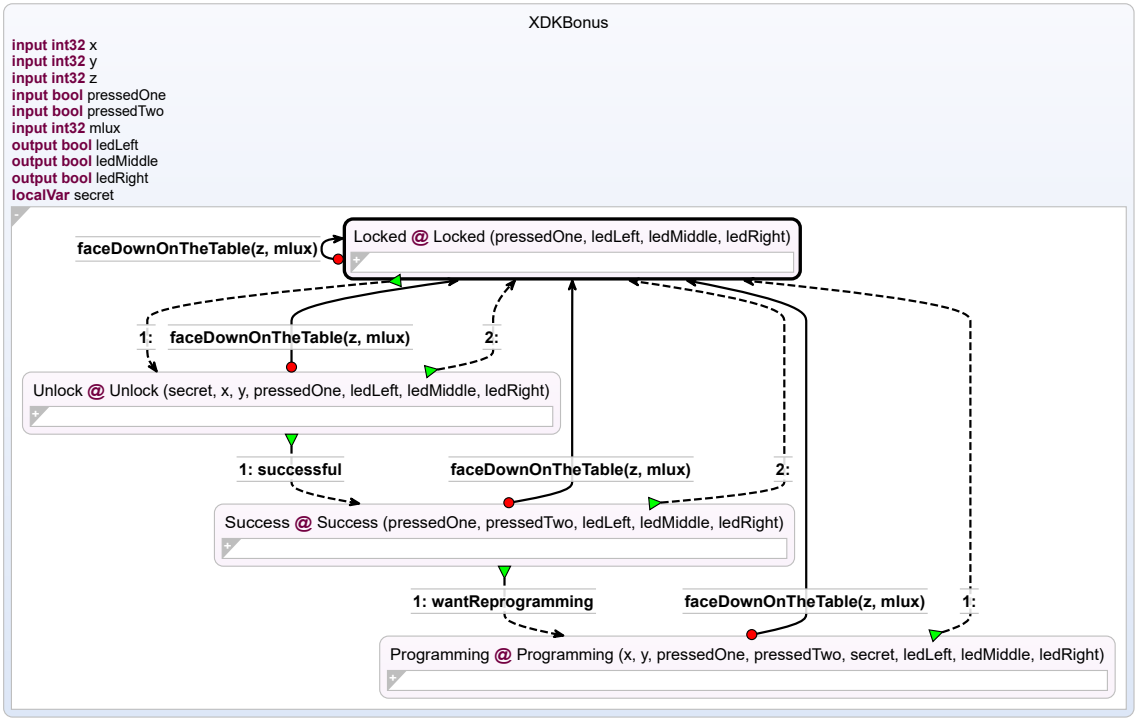


Figure A.5. Visualization of a virtual safe lock for the expert survey.

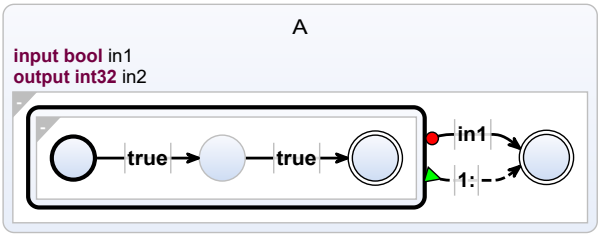


Figure A.6. Hierarchical abort construct visualization for the expert survey.

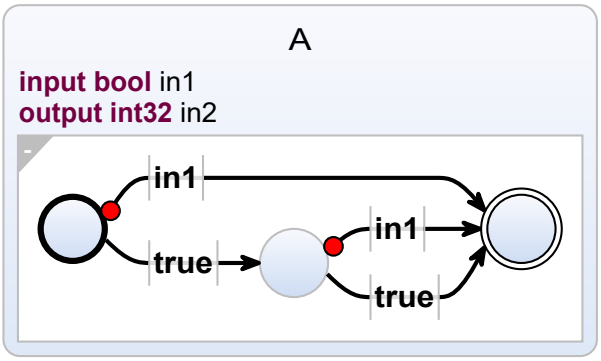


Figure A.7. Flattened abort construct visualization for the expert survey.

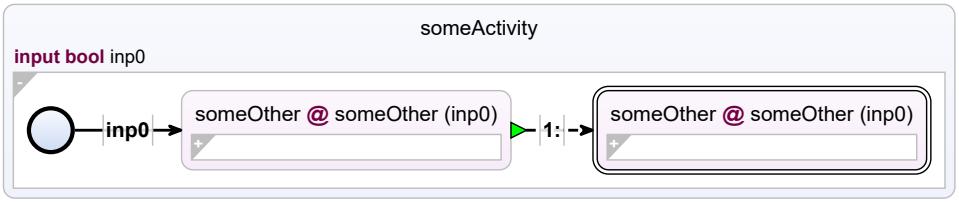


Figure A.8. Hierarchical run statements visualization for the expert survey.

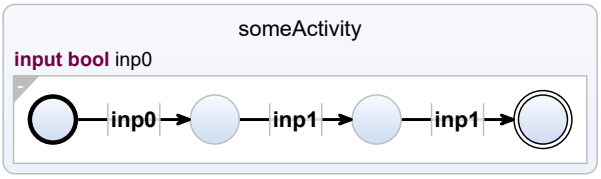


Figure A.9. Flattened run statements visualization for the expert survey.

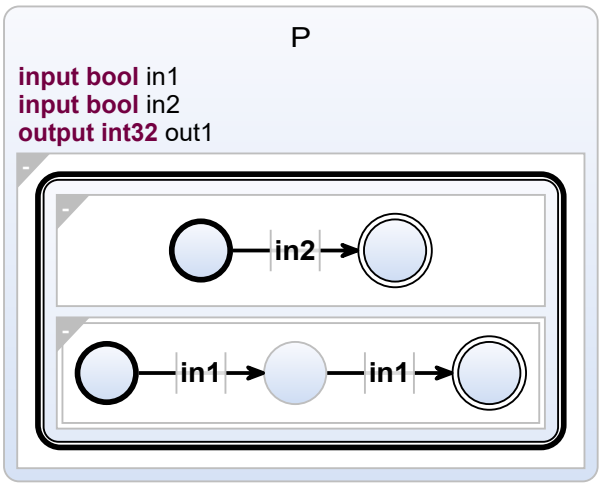


Figure A.10. Visualization of a hierarchical cobegin meeting the weak-abort pattern for the expert survey.

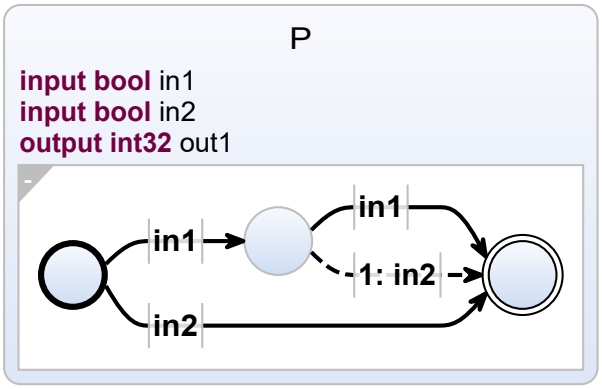


Figure A.11. Visualization of a flattened cobegin meeting the weak-abort pattern for the expert survey.

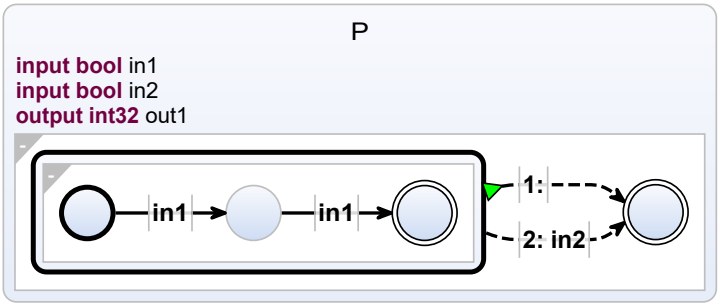


Figure A.12. Visualization of an alternative flattened cobegin meeting the weak-abort pattern for the expert survey.

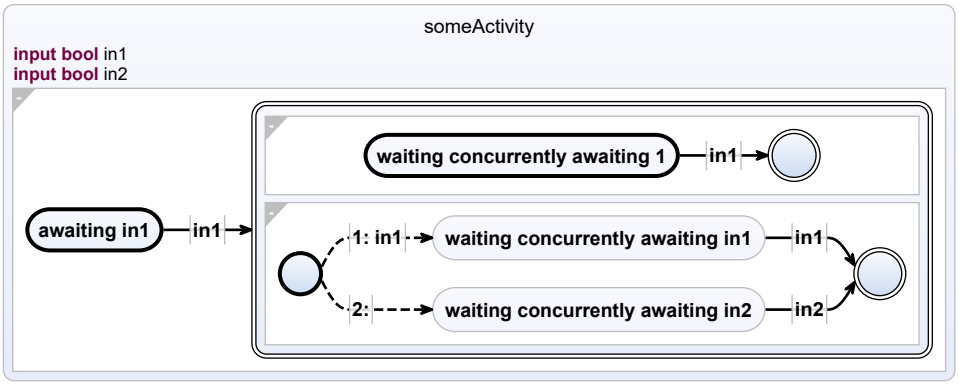


Figure A.13. Example of the state-only labeling for the expert survey.

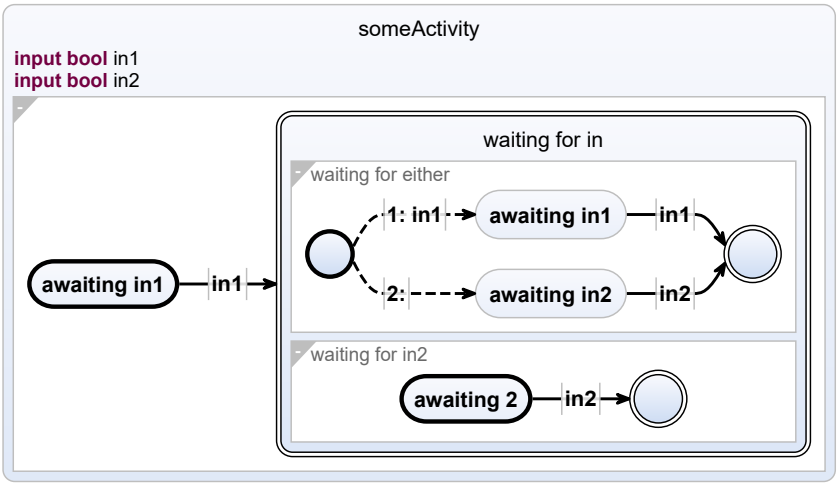


Figure A.14. Example of the advanced labeling for the expert survey.

Bibliography

[21] *Sccharts webpage*. Apr. 2021. URL: https://rtsys.informatik.uni-kiel.de/confluence/display/KIELER/Syntax.

[And19] Lewe Andersen. "Dataflow and state maschine extraction from c/c++ code". https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/lan-mt.pdf. Master thesis. Kiel University, Department of Computer Science, Dec. 2019.

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

[BD04] Marat Boshernitsan and Michael Sean Downes. *Visual programming languages: a survey*. Citeseer, 2004.

[Ber00] Gérard Berry. "The foundations of Esterel". In: *Proof, Language, and Interaction: Essays in Honour of Robin Milner*. Cambridge, MA, USA: MIT Press, 2000, pp. 425–454. ISBN: 0-262-16188-5.

[CPH+87] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. "Lustre: a declarative language for programming synchronous systems". In: *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'87)*. Munich, Germany: ACM, 1987, pp. 178–188.

[EJL+03] Johan Eker, Jörn W. Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Stephen Neuendorffer, Sonia Sachs, and Yuhong Xiong. "Taming heterogeneity—the Ptolemy approach". In: *Proceedings of the IEEE* 91.1 (Jan. 2003), pp. 127–144. ISSN: 0018-9219. DOI: 10.1109/JPROC.2002.805829.

[FH09] Hauke Fuhrmann and Reinhard von Hanxleden. *The Kiel Integrated Environment for Layout for the Eclipse RichClientPlatform (KIELER) Homepage*. https://www.rtsys.informatik.uni-kiel.de/en/research/kieler. 2009.

[FVC+13] Jobelle Firme, Nicolás Valera, Yunus Canemre, S. Burchill, and Beenish Khurshid. "Programming language families". In: 2013.

[FWW+13] Florian Fittkau, Jan Waller, Christian Wulf, and Wilhelm Hasselbring. "Live trace visualization for comprehending large software landscapes: The ExplorViz approach". In: *Proceedings of the 1st IEEE International Working Conference on Software Visualization (VISSOFT'13)*. Sept. 2013, pp. 1–4. DOI: 10.1109/VISSOFT.2013.6650536.

[GG18] Friedrich Gretz and Franz-Josef Grosch. "Blech, imperative synchronous programming!" In: *2018 Forum on Specification Design Languages (FDL)*. Sept. 2018, pp. 5–16. DOI: 10.1109/FDL.2018.8524036.

[GME05] Denis Gračanin, Krešimir Matković, and Mohamed Eltoweissy. “Software visualization”. In: *Innovations in Systems and Software Engineering* 1.2 (2005), pp. 221–230.

[Hal98] Nicolas Halbwachs. “Synchronous programming of reactive systems, a tutorial and commented bibliography”. In: *Tenth International Conference on Computer-Aided Verification, CAV '98*. Vancouver (B.C.): LNCS 1427, Springer Verlag, June 1998.

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

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

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

[LR01] Avraham Leff and James T. Rayfield. “Web-application development using the model/view/controller design pattern”. In: *Proceedings of the 5th IEEE International Conference on Enterprise Distributed Object Computing*. EDOC '01. USA: IEEE Computer Society, 2001, p. 118. ISBN: 076951345X.

[Mot17] Christian Motika. *Scchart—language and interactive incremental implementation*. Kiel Computer Science Series 2017/2. Dissertation, Faculty of Engineering, Christian-Albrechts-Universität zu Kiel. Department of Computer Science, 2017.

[MW97] Dieter Maurer and Reinhard Wilhelm. *Übersetzerbau - Theorie, Konstruktion, Generierung*. Springer, 1997. ISBN: ISBN 3-540-61692-6.

[Rüe11] Ulf Rüegg. “Interactive transformations for visual models”. Bachelor Thesis. Kiel University, Department of Computer Science, Mar. 2011.

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

[SIR15] Francisco Sant' Anna, Roberto Ierusalimschy, and Noemi Rodriguez. “Structured synchronous reactive programming with céu”. In: *Proceedings of the 14th International Conference on Modularity*. MODULARITY 2015. Fort Collins, CO, USA: Association for Computing Machinery, 2015, pp. 29–40. ISBN: 9781450332491. DOI: 10.1145/2724525.2724571. URL: https://doi.org/10.1145/2724525.2724571.

[SLH16] Steven Smyth, Stephan Lenga, and Reinhard von Hanxleden. “Model extraction for legacy C programs with SCCharts”. In: *Proceedings of the 7th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA ’16), Doctoral Symposium*. Vol. 74. Electronic Communications of the EASST. With accompanying poster. Corfu, Greece, Oct. 2016. DOI: 10.14279/tuj.eceasst.74.1044.

[Smi09] Chris Smith. *Programming f#: a comprehensive guide for writing simple code to solve complex problems*. " O’Reilly Media, Inc.", 2009.

[SSH19] Steven Smyth, Alexander Schulz-Rosengarten, and Reinhard von Hanxleden. “Practical causality handling for synchronous languages”. In: *Proc. Design, Automation and Test in Europe Conference (DATE ’19)*. Florence, Italy: IEEE, Mar. 2019.

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

[Wan19] Jiacun Wang. *Formal methods in computer science*. CRC Press, 2019.