

# Structure-Based Editing for SCCharts

Felix Jöhnk

Master thesis  
May 2023

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

Advised by  
M.Sc. Sören Domrös



### **Selbstständigkeitserklärung**

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

Kiel,

  
\_\_\_\_\_



# Abstract

Everyone wants to make programming simpler. To achieve this, there are numerous tools like auto-completion and highlighting of code in an editor. Another way of achieving this is to have a diagram based representation of the code alongside the textual representation.

While the visual representation gives a clearer understanding of the behavior of the program, we need to adjust the textual representation in order to change the visual representation. Since we need to know where to adjust the program in the textual representation, we need to traverse between visual representation and textual representation in order to change the behavior of the program.

Introducing structure-based editing can solve this problem. Structure-based editing refers to the idea of transitioning from a structurally correct model to a new structurally correct model. This concept allows to abstract from the graph and directly edit the underlying model.

This thesis starts here and implements a way in the KIELER framework for the interaction with a graphical representation of programs to adjust that same program in the process, realizing the Model-Driven Engineering (MDE) principle. To achieve this, a context menu is added and the entries, which are presented in the menu, are defined by the graphical item under the mouse when opening the context menu. The main goal is to make the context menu as interactive as possible, meaning the number of clicks and changes from mouse to keyboard should be minimized.

## **Acknowledgments**

Thanks are going to the entire group of Real-time and Embedded Systems for helping with any questions regarding the project and giving new ideas regarding the solution of problems. Special thanks go to Sören Domrös who supported me throughout the entire thesis and was always ready to help with constructive feedback.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Problem Statement	2
1.2	Outline	4
<b>2</b>	<b>Preliminaries</b>	<b>5</b>
2.1	KIELER-Framework	5
2.2	Language Server Protocol (LSP)	6
2.3	Sprotty	7
2.4	Webviews in VS Code	7
2.5	Semantic filtering and tagging API for KIELER	9
2.6	Document Object Model (DOM) Manipulation	9
2.7	Sequentially Constructive Statecharts	10
2.8	Structure-Based Programming	11
2.9	Structure-Based Programming for Statecharts	11
2.9.1	Other Actions for Sequentially Constructive Statecharts (SCCharts)	13
<b>3</b>	<b>Related Work</b>	<b>15</b>
3.1	Deuce	15
3.2	Intentional Layout	15
3.3	Graphical Language Server Platform (GLSP)	16
3.4	Eclipse Graphical Editing Framework	16
3.5	The bigER Tool	16
3.6	UML Modeling Tools	18
3.6.1	Class Diagrams	18
3.6.2	Visual Paradigm	19
<b>4</b>	<b>Structure-Based Actions for SCCharts</b>	<b>21</b>
4.1	Successor State	21
4.2	Hierarchical States	22
4.3	Initial State	22
4.4	Toggle Final	23
4.5	Add Transition	23
4.6	Change Source	24
4.7	Change Target	24
4.8	Weak, Final and Aborting Transition	25
4.9	Deletion	25
4.9.1	States	25
4.9.2	Transitions	25
4.9.3	Regions	26

## Contents

<b>5</b>	<b>User Story</b>	<b>27</b>
5.1	Successor State . . . . .	28
5.2	Concurrent State . . . . .	28
5.3	Add Transition . . . . .	28
5.4	Initial State . . . . .	29
5.5	Toggle Final . . . . .	29
5.6	Change Source . . . . .	29
5.7	Change Target . . . . .	30
5.8	Weak, Final and Aborting Transition . . . . .	31
5.9	Deletion . . . . .	31
<b>6</b>	<b>Implementation</b>	<b>33</b>
6.1	Client Interface . . . . .	33
6.1.1	ContextMenuProvider . . . . .	35
6.1.2	Arrow Drawing . . . . .	37
6.1.3	Action Creation . . . . .	37
6.2	Server Side . . . . .	37
6.2.1	Server Action Handler . . . . .	38
6.2.2	Structural Editing Server Extension . . . . .	38
<b>7</b>	<b>Evaluation</b>	<b>41</b>
7.1	Clicks for Structure Changes . . . . .	41
7.2	Test . . . . .	41
7.2.1	ABRO . . . . .	43
7.2.2	Modulo . . . . .	43
7.2.3	Elevator . . . . .	43
7.3	Test Results . . . . .	45
7.4	Comparing Structure-Based editing to Traditional Graph editing . . . . .	49
<b>8</b>	<b>Conclusion</b>	<b>51</b>
8.1	Summary . . . . .	51
8.2	Future Work . . . . .	51
	<b>Bibliography</b>	<b>55</b>



# List of Figures

1.1	Using a graph to display transitions. . . . .	1
1.2	KIELER software displaying the text and generated graph together. . . . .	3
1.3	Changing a SCChart in text and graph. . . . .	3
2.1	KLighD Diagram generation . . . . .	5
2.2	Interaction between Sprotty, VS Code and the Language Server. . . . .	6
2.3	Difference between working with LSP and without it. . . . .	7
2.4	Sprotty architecture . . . . .	8
2.5	VS Code Extension . . . . .	9
2.6	Textual and graphical representation of an SCChart. . . . .	10
2.7	Generic editing schemata [Pro08] . . . . .	12
3.1	Deuce Structure-Based editing . . . . .	15
3.2	User interactions in GLSP . . . . .	17
3.3	The BigER tool . . . . .	18
4.1	Structured change produced by add successor action. . . . .	21
4.2	Structured change produced by add successor action. . . . .	22
4.3	Problematic to add concurrent region waitB. . . . .	23
4.4	Modulo counter . . . . .	24
4.5	Changing the source of an transition. . . . .	24
4.6	Changing the target of an transition. . . . .	25
5.1	Contextmenus for states, transitions and regions. . . . .	27
5.2	Context menu for Add successor state action. . . . .	28
5.3	Contextmenu for Add region and add concurrent region options. . . . .	28
5.4	User storys for adding transitions. . . . .	29
5.5	User stories for changing the state type. . . . .	30
5.6	Changing the source using the arrow functionality. . . . .	30
5.7	Changing the source using the arrow functionality. . . . .	30
6.1	Steps performed for structure-based editing. . . . .	33
6.2	Definition of inputs for the Add Transition action. . . . .	36
6.3	Implemented interfaces and corresponding SCChart-specific components. Other languages can be added by updating the service loader and implementing similar classes to the components on the right. . . . .	39
7.1	ABRO as graph . . . . .	44
7.2	Example graph for modulo test. . . . .	46
7.3	Elevator as graph . . . . .	47
7.4	Graph solution for modulo test from text editing user. . . . .	48
7.5	Automatic resizing and placement of nodes in a place efficient way. . . . .	49

List of Figures

7.6	diagrams.net application with the text and shape on the right selected. . . . .	50
8.1	Simulating a SCChart . . . . .	52
8.2	Interactive simulation . . . . .	53
8.3	Obtaining all possible input paths to state done. . . . .	54

# List of Abbreviations

<i>DSL</i>	Domain Specific Language
<i>MDE</i>	Model-Driven Engineering
<i>SCChart</i>	Sequentially Constructive Statechart
<i>KIELER</i>	Kiel Integrated Environment for Layout Eclipse Rich Client
<i>IDE</i>	Integrated Development Environment
<i>ELK</i>	Eclipse Layout Kernel
<i>LSP</i>	Language Server Protocol
<i>SVG</i>	Scalable Vector Graphic
<i>GEF</i>	Eclipse Graphical Editing Framework
<i>GLSP</i>	Graphical Language Server Platform
<i>DOM</i>	Document Object Model
<i>MVC</i>	Model View Controller
<i>UML</i>	Unified Modeling Language
<i>ELK</i>	Eclipse Layout Kernel



# Introduction

The field of programming requires a vast amount of knowledge regarding the programming languages used. The fact that one needs to apply knowledge from concepts and language to a specific topic is one of the key points why learning and understanding code at the beginning can be problematic [Tsa18]. In graphical programming the programming languages syntax is hidden by the graph and, therefore, only the knowledge of the topic and some broader concepts and how the graph syntax works are necessary. Additionally, if altering the program through interactions with the graph is possible, the language syntax can be hidden for programming as well. Since the graph syntax is ideally simple, even people without a lot of knowledge in programming can, without too much effort, understand graphs while having no understanding of the textual representation. A benefit of a graphical representation is that the relationships between states, objects or classes are simpler to understand [NK98]. An example is given in Listing 1.1 and Figure 1.1. In the example the desired information is what states are connected to the state *normal*. In the graph we directly see all three states that are connected to it. In the text we need to scan every transition after every state in order to know, which states are connected. While the graphical representation has its benefits, the textual representation is useful as well. One of the benefits of editing the textual representation is that the user needs less mouse interactions, which can improve productivity [ACL+17]. This means that textual and graphical modeling approaches can improve different aspects of designing a program.

*Hybrid* (or *blended*) modeling is referring to modeling that utilizes textual representations as well as a graphical representation. The main goal is that the user is able to interact with the model in more ways to increase the productivity [CTV+19]. The model is then abstracted into, for example, a graph that may be displayed alongside the text file.

Similar to hybrid modeling, *modeling pragmatics* formulates the idea that any practical interactions with the model is addressed [HLF+22]. This includes creating views that represent the model graphically, browsing the graphical model and simulating the model. Ideally, the created tools help in documenting and understanding changes during development without the need to produce models by hand. The idea of modeling pragmatics is closely related to the Model View Controller (MVC) pattern.

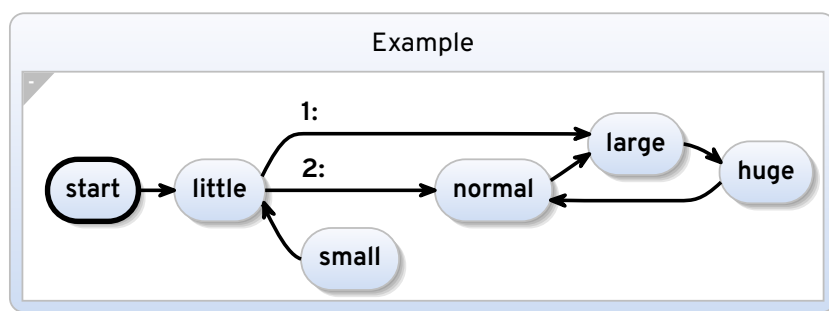


Figure 1.1. Using a graph to display transitions.

## 1. Introduction

```
1 scchart Example {  
2   initial state start  
3   go to little  
4  
5   state little  
6   go to large  
7   go to normal  
8  
9   state small  
10  go to little  
11  
12  state normal  
13  go to large  
14  
15  state large  
16  go to huge  
17  
18  state huge  
19  go to normal  
20 }
```

**Listing 1.1.** Using text to display transitions.

Both approaches abstract from the model to generate different views and interaction possibilities. This can result in extension for VS Code like the `klighd-vscode` extension<sup>1</sup>, which is displayed in Figure 1.2.

An option, for interacting with a diagram, is to utilize the structure-based programming principle [PH07]. This allows the user to create and adjust the program from a given diagram. The idea is to adjust the program in such a way that any change to the structure results in a structurally correct program. This fits the idea of modeling pragmatics allowing the user to generate diagrams from a textual representation and changing the meaning of a program through an interaction possibility in the graphical interface.

### 1.1 Problem Statement

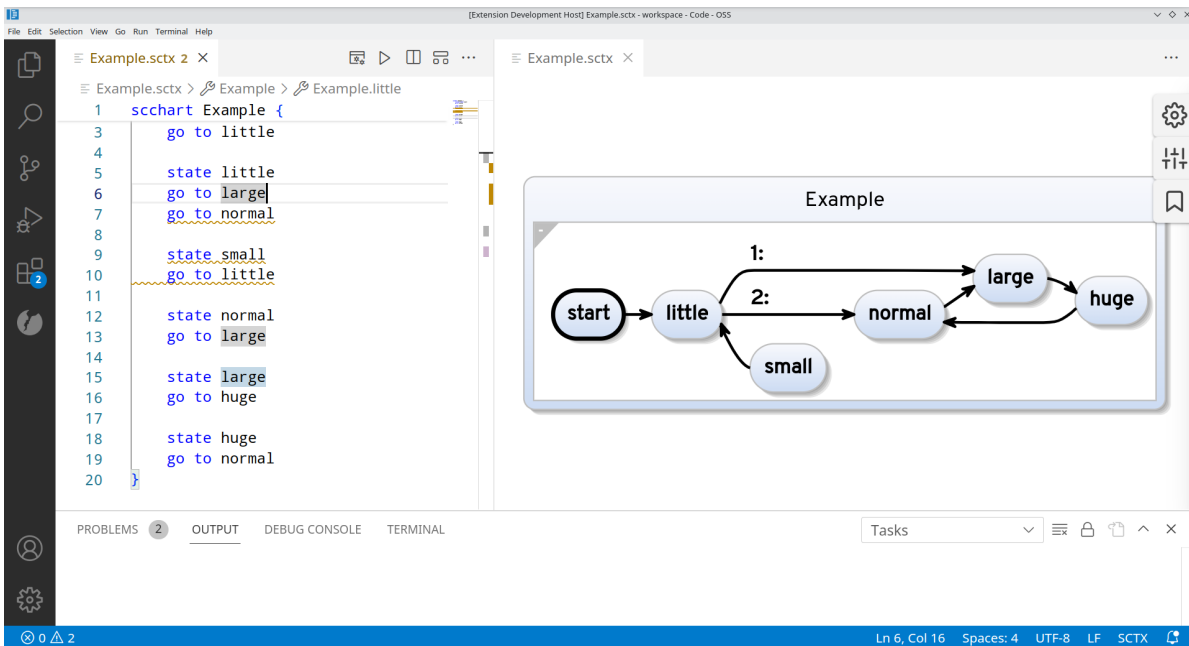
The goal of this thesis is to introduce an interaction possibility in the KIELER framework for `SCChart` diagrams that can alter the meaning of the program. The previous behavior was to adjust the text file directly, as displayed in Figure 1.3a. Figure 1.3b shows the graph prior to any changes while Figure 1.3c is displaying the graph after the change.

The new interaction possibility is realized using the structure-based editing approach. The first step during creation of this feature is to evaluate, which structure-based actions can be performed. Structure-based actions refer to any structure change to a diagram that can be performed by the user. The next necessary step is to introduce an interaction interface in the front-end. Since the KIELER framework was migrated to web-technologies by Fricke [Fri21], I decided to utilize a context menu, which is a common interaction possibility in web interfaces. This context menu should have an interface, which can be used not only by the structure-based editing functionality but also by other

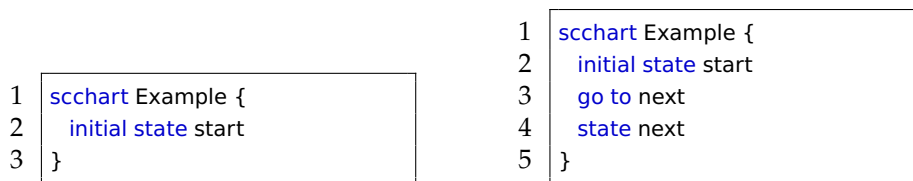
---

<sup>1</sup><https://github.com/kieler/klighd-vscode>

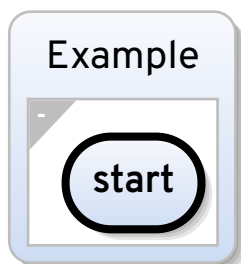
## 1.1. Problem Statement



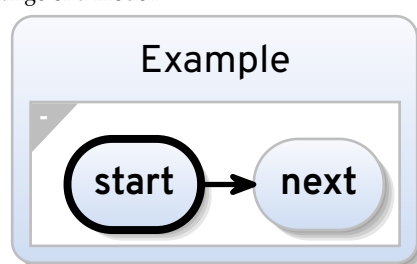
**Figure 1.2.** KIELER software displaying the text and generated graph together.



(a) Textual change of a model.



(b) Prior to any changes.



(c) Result from both changes.

**Figure 1.3.** Changing a SCChart in text and graph.

projects that want some interactive features. For example new languages supporting structure-based editing. Lastly a server adjustment is necessary to handle the structure-based actions and update the diagram.

## 1. Introduction

### 1.2 Outline

Chapter 2 explains technologies that are part of the implementation as well as main concepts. The related work in Chapter 3 shows different approaches utilizing the hybrid programming principles. Chapter 4 describes the structure-based actions, which can be done for SCCharts. Directly after that the Chapter 5 shows the user story for every structural change. Chapter 6 explains the approach to implement the ideas from Chapter 4 and Chapter 5. Chapter 7 considers user experience and click counts while using the structure-based editing approach for an evaluation and Chapter 8 finally shows the achieved goals and gives information on future work.



# Preliminaries

This chapter explains the main technologies and concepts. Starting with an overview of the KIELER framework and its components Sprotty, LSP and the VS Code extension, which is the where the interaction with the user happens. Continuing with a brief introduction into the SCChart syntax. Lastly, an explanation for the idea of structure-based programming for statecharts is given.

## 2.1 KIELER-Framework

The goal of the Kiel Integrated Enviroment for Layout Eclipse Rich Client (KIELER)-Project is to apply the concept of modeling pragmatics to create an Integrated Development Environment (IDE) for MDE [HLF+22]. Since modeling pragmatics requires to abstract from a model to introduce different views, the main truth is stored in the textual representation of a Domain Specific Language (DSL). The steps to generate a diagram, from a textual representation, are depicted in Figure 2.1.

Parsing the textual model into a Java model of a specific DSL is the first step. The synthesis takes the Java model and performs a transformation into a *KGraph*. The *KGraph* is a model storing information for the visualization. Formulating a layout problem for the *KGraph* and solving it, using the Eclipse Layout Kernel (ELK), results in the position and size of nodes. The combination of the information from the layout with the information from the *KGraph* results in the desired diagram, which the front-end is displaying to the user. The larger dotted lines are the additions I propose in this work.

Initially KIELER was integrated in Eclipse and was then migrated in the context of web-technologies by Domrös [Sör18] and Rentz [Ren18] to Theia resulting in KEITH. KEITH uses the Sprotty framework to display diagrams in a web container. The work of Fricke [Fri21] then integrated the view of the model into VS Code which resulted in the current state-of-the-art program depicted in Figure 1.2.

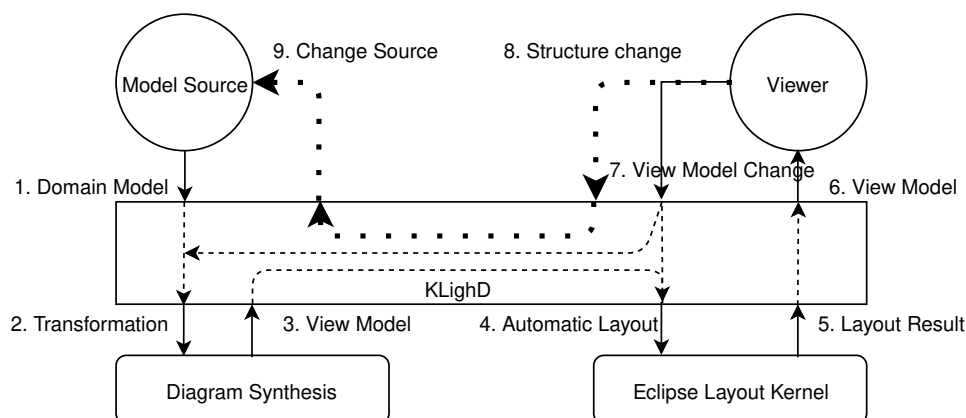


Figure 2.1. KLightD Diagram generation

## 2. Preliminaries

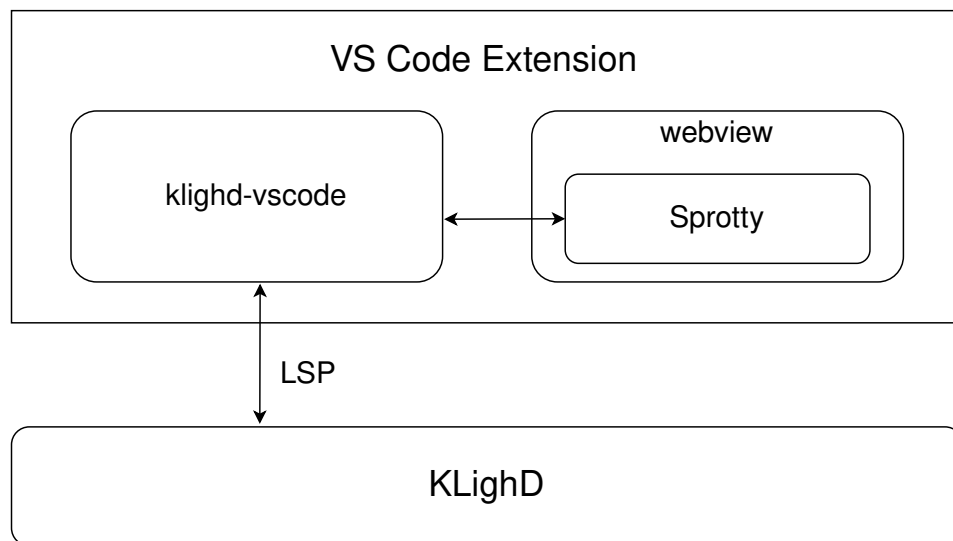


Figure 2.2. Interaction between Sprotty, VS Code and the Language Server.

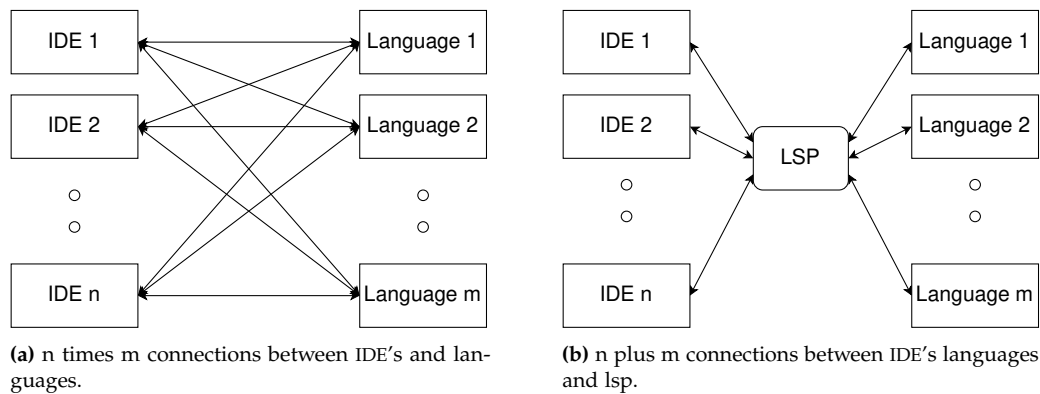
Displaying a diagram with the KIELER framework requires a front-end in which a diagram can be displayed and a communication interface between them. LSP functions as a connection point between the back end and the front-end while the displaying of a diagram is currently done using Sprotty, as depicted in Figure 2.2. While KLightD only communicates with klighd-vscode through the use of LSP the information needs to be forwarded to the webview in which the diagram is displayed. Also Sprotty may provide updates or interactions to the graphical model which are then forwarded to KLightD.

## 2.2 LSP

Most IDE's support a variety of features depending on the specific language. Those features range from finding a specific text to auto-completion and even producing graphs depicting the behavior for the currently open program. Those features need to be coded into the IDE and, therefore, every new IDE needs its own extensions for each language, as depicted in Figure 2.3a. The problem with this approach is that every IDE needs its own extension for every language resulting in  $n * m$  extensions [KPE16]. This problem can be solved using LSP<sup>1</sup>.

Reducing the number of extensions with LSP works by connecting a client to LSP, which can be an IDE, as well as a language server who provides specific features to LSP. In doing this we can use existing language server implementations, who already provide some behavior, through LSP to a new IDE. Therefore, every IDE can use the same language server. This results in  $n$  extensions from the IDEs to LSP and  $m$  extensions from LSP to the servers as depicted in Figure 2.3b. In total, we can reduce the number of extensions to  $n + m$  by using LSP.

<sup>1</sup><https://microsoft.github.io/language-server-protocol/overviews/lsp/overview/>



**Figure 2.3.** Difference between working with LSP and without it.

## 2.3 Sprotty

Developed by Typefox, Sprotty<sup>2</sup> functions as a web-based graphics framework for diagrams. It uses Scalable Vector Graphics (SVGs) to display graphs and while it is possible to use without a server, it supports language servers for more sophisticated diagrams. The main goal of Sprotty is to provide graphical feedback and interaction possibilities to the diagram. For this reason Sprotty allows browsing through the displayed model by dragging the content around, can handle changes to the text model and update the diagram using animations.

The Sprotty architecture is based on a unidirectional cyclic event flow depicted in Figure 2.4. Actions are the driving factor in changing the diagram. The viewer as well as the model source can produce actions. The viewer produces actions required by the user whereas the model source is producing actions that handle updates to the source code. Every action is assigned to a specific action handler. This is done by assigning a unique kind to each action and registering the kind at the action handler. The action handlers are not used directly but rather are registered in the action dispatcher. The idea is to use the action dispatcher as a single point that can handle any action given to it. The model source is also an action handler and therefore, the communication between server and client utilizes actions as well. After retrieving a command from an action handler it is forwarded to the command stack. A command specifies the behavior of the change requested by an action. This means it takes the current model and adjusts it according to the command to return a new model.

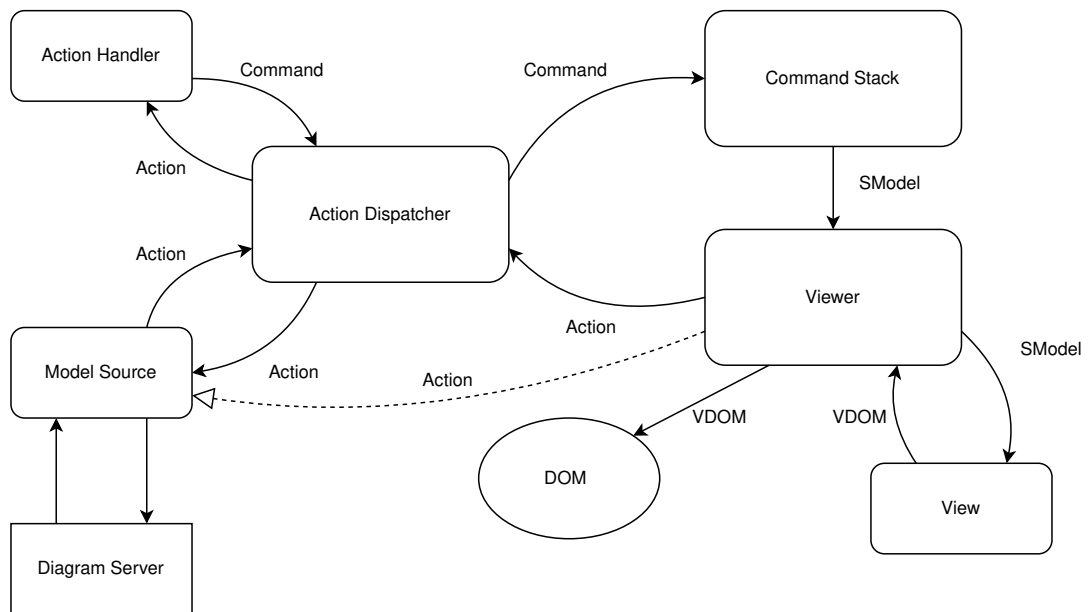
Sprotty in itself is not integrated in VS Code but is rather creating a HTML document that, when opened in a browser, displays a model. For this reason when integrating Sprotty into VS Code, Sprotty is run inside a webview, as displayed in Figure 2.2. The communication to the language server is done through the use of klighd-vscode, which communicates with the language server using LSP.

## 2.4 Webviews in VS Code

VS Code is an open source IDE mainly developed by Microsoft. Whereas VS Code provides code refactoring, navigation, formatting besides other features for JavaScript out of the box, other languages have no such support build in. For those languages new extensions can be created. Those extensions can be from Microsoft or the community who uses VS Code.

<sup>2</sup><https://www.typefox.io/blog/sprotty-a-web-based-diagramming-framework>

## 2. Preliminaries



**Figure 2.4.** Sprotty architecture

The functionality of extensions is not limited to specific features but can range from language extensions over auto-completion and even producing new interfaces. The features need to be implemented in an extension in VS Code, which are written completely in Node.js or JavaScript. Those extensions can, however, communicate with services outside VS Code, which provide some behavior.

The interaction between extensions and VS Code is realized using the VS Code API<sup>3</sup>. The api offers a variety of options including code highlighting and adding new items to menus. An example for code highlighting is depicted in Figure 2.5 on the left. The red underlying text indicates the error that the state `waitA` has no interior behavior.

Part of the VS Code API is the Webview API that allows the user to define fully customizable views in VS Code. A webview in that sense is closely related to iframes in traditional browsers. It can render almost any HTML content and, therefore, allows the author of an extension to produce any view, which would be possible to display in a browser. In Figure 2.5 on the right is a diagram displaying the code on the left. This view is realized using the webview API.

Since VS Code runs on Electron, which is used to provide cross-platform desktop applications and uses chromium to display interfaces, it can also be run completely in the browser<sup>4</sup>. There are some limitations for browsers which, therefore, effect VS Code in the browser version. For example, if the browser in use is not supporting the File System Access API, all files need to be uploaded. This is necessary if the user wants functionality like code browsing or refactoring across files.

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

<sup>4</sup><https://vscode.dev/>

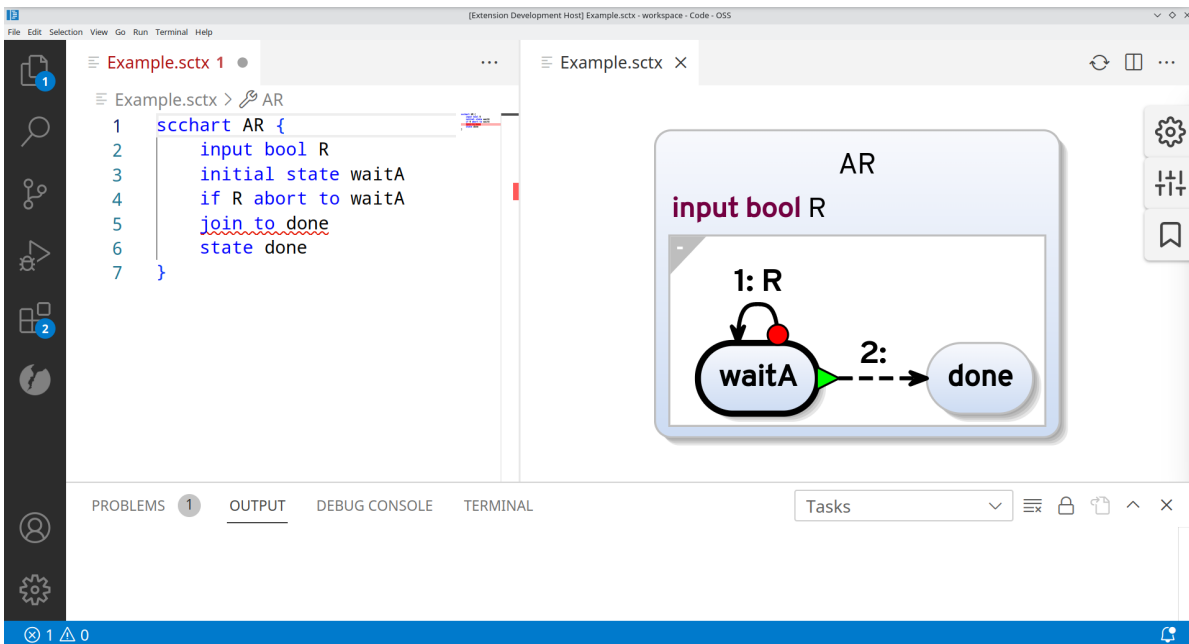


Figure 2.5. VS Code Extension

## 2.5 Semantic filtering and tagging API for KIELER

Kasperowski introduced the idea of semantic filtering and tagging into KIELER<sup>5</sup>. The idea is that tags can be used to distinguish between the different elements in a graph and filter the graph depending on elements. Additionally, the tags enable the user to distinguish between different elements even if they are drawn using the same SVG class. For example, it is not directly possible without tags to distinguish between regions and states for SCCharts. To achieve this tags, which can for example be strings, are added to every type of element of a model. On the client the tags can be accessed and thus enable the client to distinguish between different elements.

## 2.6 DOM Manipulation

In order to add interactive functionality, we need to add an interface, in which the user can enter information or select specific options. The context menu in VS Code is not capable of adjustments during runtime. This means that we need to add this functionality to klighd-vscode. One way to achieve this is through the use of DOM manipulation. The DOM is a model of a HTML website. It is build up in a tree structure with its root being the first HTML tag in the text. Then every tag inside the root tag is added as a child node. This happens recursively until every tag has been added in the DOM.

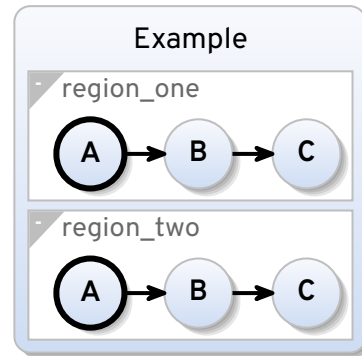
DOM Manipulation allows the user to change the DOM with JavaScript using jQuery methods [Pan14]. This includes finding elements in the DOM by id, deletion of nodes from the DOM as well as adding new nodes in the DOM and other options. This directly alters the appearance of the displayed website and, therefore, can be used to add interactive features to a website.

<sup>5</sup><https://github.com/kieler/KLighD/pull/137>

## 2. Preliminaries

```
1 scchart Example {  
2   region region_one {  
3     initial state A  
4     go to B  
5  
6     state B  
7     go to C  
8  
9     state C  
10  }  
11  region region_two {  
12    initial state A  
13    go to B  
14  
15    state B  
16    go to C  
17  
18    state C  
19  }  
20 }
```

(a) Textual representation



(b) Graphical representation

Figure 2.6. Textual and graphical representation of an SCChart.

## 2.7 Sequentially Constructive Statecharts

The KIELER framework supports drawing diagrams for programs written in the SCCharts syntax. SCCharts are made for safety-critical reactive systems [HDM+14]. Since SCCharts are based on statecharts it is also a synchronous language. The main difference to statecharts is that sequential constructivity can be provided using the synchronous model of computation. The concept of SCCharts is closely related to state machines.

An example SCChart can be seen in Figure 2.6. In Figure 2.6a is the textual representation and in Figure 2.6b the graphical representation. The keyword `region` followed by an id on the left indicates a section of code, which is executed concurrently to all other regions on the same level. In the graph regions are displayed as boxes with a minus in the top left corner and the id following the minus. For example the region from line two to ten is displayed as the top box in Figure 2.6b. The keyword `state` followed by an id is seen on the right as a node with the id shown in its center. For example the state in line three is shown as the top left node A in Figure 2.6b. The keyword `go to` indicates an edge on the right-hand side. The edges source is the state under that the `go to` is located in text and the target is the node that has the id after the `go to` in the same region. For example the edge defined in line four is depicted as the edge from A to B in the top box of Figure 2.6b.

## 2.8 Structure-Based Programming

Traditionally editors work by simple line editing. This means that during the editing of a program there are multiple stages where the structure of said program is incorrect. E.g. in a Java program one opens a bracket for an if statement but never closes it while writing the behavior for the if cases.

Structure-based programming is a type of programming that forces the user to adjust the program such that the program is always structurally correct. This means for the example that it would directly create a closing bracket for the opening bracket of the if statement.

All code editors who support auto-completion have a mix of structure-based and line editing build in them since once the auto-completion is done it will result in an almost structurally correct code base. For example auto completing an *if* statement will have no condition or interior behavior, but it will have closing brackets at all points. However, to be correct we need all if statements to have a condition and an interior behavior.

## 2.9 Structure-Based Programming for Statecharts

One of the points from the research from Prochnow is how structural changes effect the layout of graphs and what actions need to be considered in order to create good looking graphs [Pro08]. Following the concept of modeling pragmatics the idea was to generate diagrams from a textual representation in order to simplify the modeling process of a programmer. As part of the work structure-based programming was implemented for statecharts in a new tool named macro editor. The goal was to simplify the process of diagram generation since in traditional graph programming new nodes require careful consideration in the layout. For this reason combining the structure-based programming approach with the automatic layout of graphs results in a graphical programming interface which does not require the user to perform any layout for new nodes or edges.

For the editing process of state charts Prochnow splits the possible editing schemata in three groups. Either we want to create, modify or delete elements of the statechart.

### Statechart Creation

To create a functional statechart the user needs to create an initial state. To do this one needs to create a state with an initial connector, which functions as the start of the program.

In our approach for SCCharts we completely ignore the creation of a new SCChart. This comes from the fact that there are several options one can implement in the beginning of a SCChart. These options range from declaring full methods, which are executed immediately, over setting new variables for input and output signals. For this reason and since we still have the textual representation open while working in the graph we omit the creation of a new SCChart.

### Add Statechart Elements

One of the main tasks in adjusting statecharts is the addition of new states. These states do not change the meaning of the statechart if they are not connected, therefore, every state must have at least one connection. This means if we want to create a new state we need to create a state and a transition. This is also displayed in Figure 2.7a. The behavior is the same in our approach for SCCharts.

## 2. Preliminaries

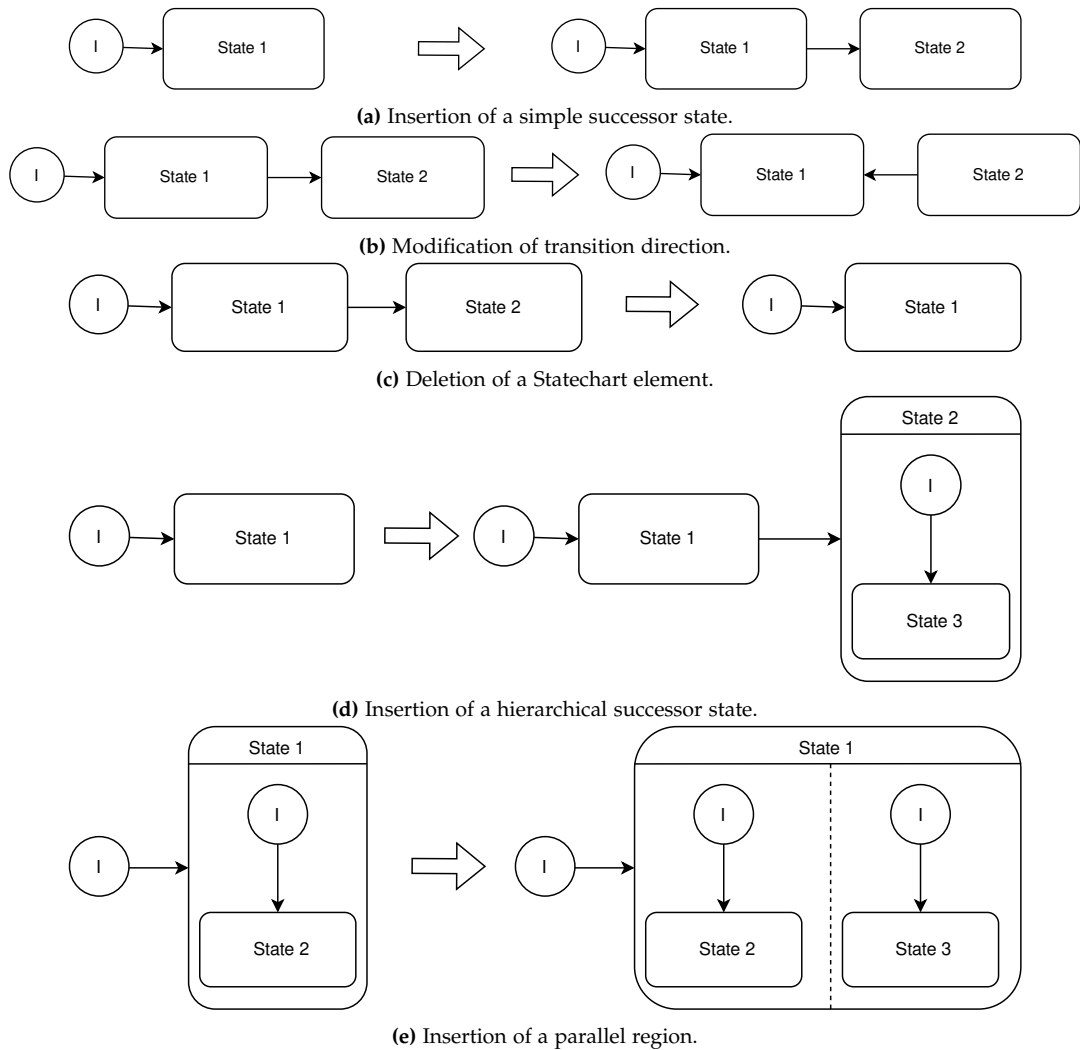


Figure 2.7. Generic editing schemata [Pro08]

### Add Hierarchical States and Parallel Regions

Adding a hierarchical state may happen in two ways. Either we want to add the first hierarchical state or we want to add another one to be executed in parallel. In both cases, we need four steps to add such a new state in our statechart. First, we need to create a region inside the state. Then we need to add an initial connector and a state to the region. Lastly, we need to add a transition from the initial connector to the state. These steps are shown for a first hierarchical state in Figure 2.7d and for multiple in Figure 2.7e.

While similar, the concept of SCCharts allows skipping some of the steps. Since SCCharts have an initial state instead of an initial connector, we can skip the creation of the transition and the initial connector. However, we need to make sure the state we create is initial. Additionally, since we already can introduce successor states I decided that changing a state to a hierarchical state is sufficient and the step depicted in Figure 2.7d is not necessary.



### Change Complexity of States

Changing the complexity of a state may cause problems. When upgrading states into hierarchical states or adding new nodes to that state the attributes should be persistent. This means that for example the name, the incoming and outgoing transitions should be the same see Figure 2.7e. This can be problematic in some modeling applications since the modeler needs to delete the old state and create a new one with the correct hierarchical nodes. While this is true for SCCharts it is simpler to adjust the model and, therefore, adding a hierarchical node doesn't cause this problem.

### Switch Transition Source and Target

The changing of a transition direction can be trouble some. This comes from the fact that one would need to delete the existing transition and create a new one with the same attributes except the source and target, which is depicted in Figure 2.7b.

While the same holds true in SCCharts the problem of reversing an edge is rather specific. This specific behavior is adjusted in this thesis to let the user change of source and target independently. In this way reversing of an edge would still take two actions. First, changing the source and then changing the target. However, in this way we can change the successor state of a state while preserving the attributes assigned to the transition.

### Delete Statechart Elements

Deletion of states can be troublesome for the user as well. The process of deleting a single state leaves all transitions from and to the state without a source or target. Since we need to achieve a structurally correct statechart we need to delete all those transitions as well. If we leave a hierarchical state without any interior behavior, which means no states, we need to change the hierarchical state to a normal state. For the deletion of states in SCCharts we need to consider the same problems.

#### 2.9.1 Other Actions for SCCharts

Since there are multiple transition and state types for SCCharts I have added more functionality that lets the user change states and transitions in more ways than the ones described in the work of Prochnow. Additionally, I added more functionality to switch between textual representation and graph in order to account for the decisions made for the creation of SCCharts.



## Related Work

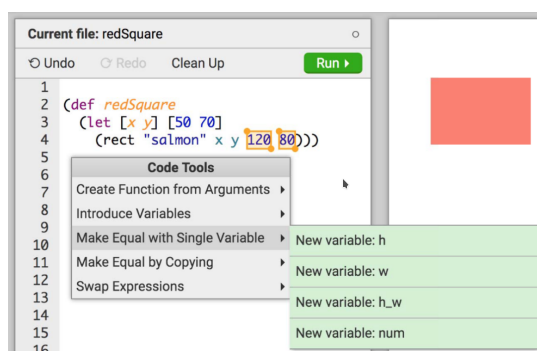
This chapter focuses on previous research in the field of graphical programming. It also explores other projects and the approaches taken to add interactivity.

### 3.1 Deuce

The idea of structure-based programming for textual code bases finds uses in applications like Deuce [HLL+18]. Deuce is a software that enables the user to structurally select elements of the code and then depending on the elements one could change the meaning. E.g. using the shift key to enter structural editing mode the user selects the two values 120 and 80 since they should be equal. After the selection is done a popup menu shows the options to adjust the meaning of the program. After hovering the option to Make Equal with Single Variable different variable names for the new variable are displayed. If hovering the option New variable: w a preview of the code change is displayed to the user, as in Figure 3.1b. As one can see the value of w is set to 80 without selection from the user. This means that the value of the constant is not selected by the user but the designer of *MakeEqual* functionality.

### 3.2 Intentional Layout

The work of Petzold et al. [Pet19; Sch19] proposed a way of intentional positioning nodes in a given layout introducing new interactive features to change a visual model from within the diagram view of the KIELER framework and not changing the text directly. This, however, did not change the meaning of the program but simply positioned nodes according to the defined constraints and changed the textual representation so that the constraints are persistent.



(a) Deuce structural change of values[HLL+18].

```
(def redSquare
  (let [x y] [100 70]
    (let w 80
      (rect "salmon" x y w w))))
```

(b) After structural change using the new variable w option[HLL+18].

Figure 3.1. Deuce Structure-Based editing

### 3. Related Work

#### 3.3 GLSP

GLSP adds new features to LSP to build a framework that enables users to build web based IDEs for MDE<sup>1</sup>. The goal is to enable the building of custom diagram editors based on web-technologies.

The difference to the KIELER framework is that the diagram is created by the user placing nodes and edges in a diagram and not creating the diagram automatically from a source file. This means that the GLSP server needs to provide a way of loading and storing models, transform a given model into a graphical representation and options to edit the model. The KIELER framework in contrast requires source file from the user that is synthesized into a view model, which then is used to create a layout. The editing of a model in the KIELER framework, therefore, was not implemented yet since the editing of the model is done in the source file and not in the displayed graph.

In the thesis of Forstner [For22] a VS Code integration was realized. After the transition to VS Code the commands could be executed in different ways. First one could use the VS Code command panel as shown in Figure 3.2a. Secondly one could use the integrated VS Code context menu for some actions like centering as seen in Figure 3.2b. The problem Forstner faced were the limitations of VS Code. One could only add new items to the context menu in VS Code and those items would always be displayed for all languages. They also noticed that a context menu, which is part of the container displayed inside the webview, could be used instead. This context menu would be customizable. For that thesis this was, however, not done and proposed as a future feature. In this thesis we faced the similar problem of adjusting the context menu and, therefore, decided to integrate the context menu inside the webview.

In a later version, a panel as well as a context menu was added<sup>2</sup>, which were implemented using DOM manipulation. The resulting menus allow the server to support language specific menu entries, as depicted in Figure 3.2.

#### 3.4 Eclipse Graphical Editing Framework

Another framework that allows the user to create graphical editors and views like xmind<sup>3</sup> and WindowBuilder<sup>4</sup> is the Eclipse Graphical Editing Framework (GEF) [RWC11]. The typical application has features like a panel, in which the model items are displayed. The applications, however, are not integrated in web-technologies but rather can be run integrated in Eclipse or as a standalone application. Also the idea is not to generate a graphical layout from a source file like the KIELER framework but rather similar to GLSP create views, which can be adjusted.

#### 3.5 The bigER Tool

The bigER Tool is developed by Glaser and Bork and is used for entity relationship modeling [GB21]. The goal of the tool is to provide hybrid editing functionality for entity relationship diagrams. It achieves this by using a similar structure to the KIELER framework. The software uses Sprotty to display the diagrams while the diagrams are created using a language server. The communication between the server and the client is using LSP. In common with the KIELER framework is also that the main truth is the code base meaning the diagram is generated from code. The framework also allows

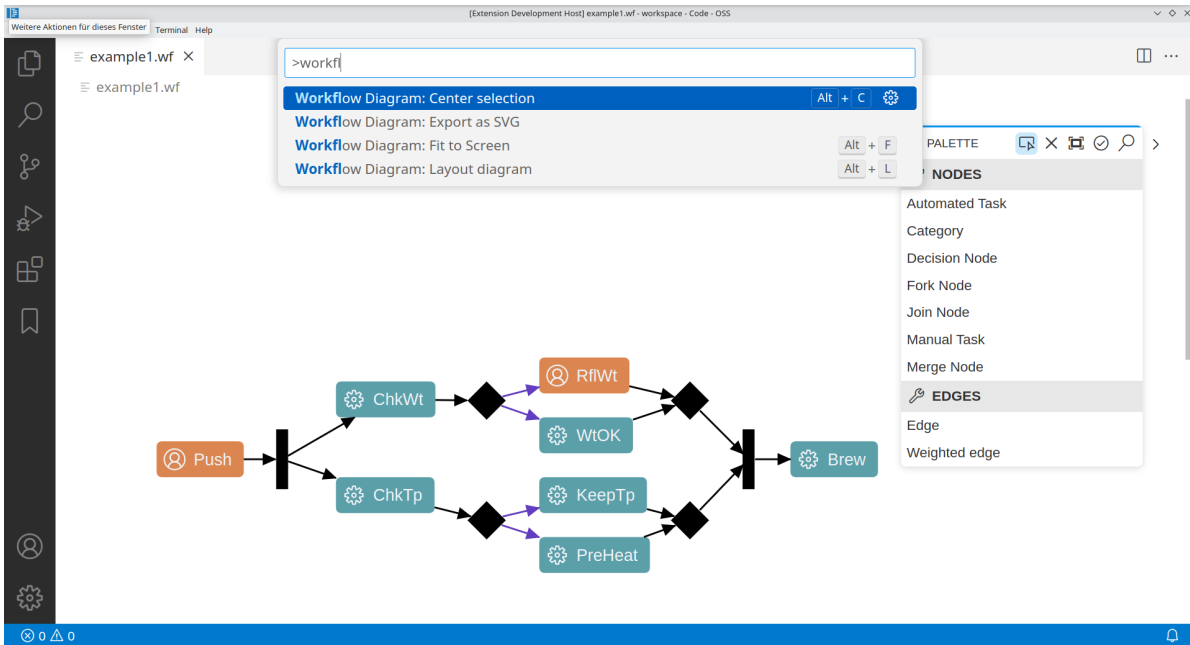
---

<sup>1</sup><https://www.eclipse.org/glsp/>

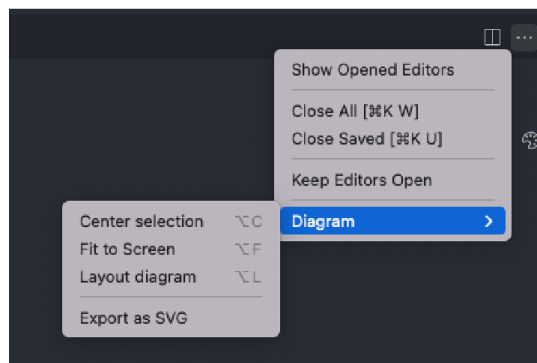
<sup>2</sup><https://www.eclipse.org/glsp/documentation/ui-extensions/>

<sup>3</sup><https://xmind.app/>

<sup>4</sup><https://www.eclipse.org/windowbuilder/>



(a) VS Code Command Panel containing the contributed GLSP Commands.



(b) VS Code menu contributions in context menu[For22].

**Figure 3.2.** User interactions in GLSP

### 3. Related Work

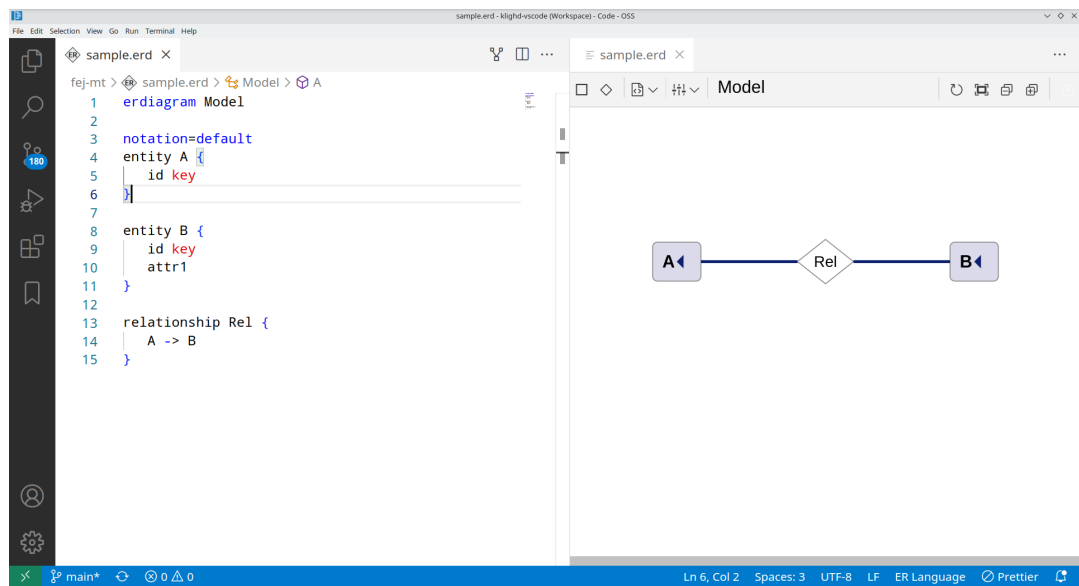


Figure 3.3. The BigER tool

the user to interact with the diagram and adjust the code base from the diagram view. This includes adding new attributes to entities, creating new entities, renaming or deleting entities and creating new relations. A depiction of the extension at work is given in Figure 3.3.

While this thesis also enables hybrid modeling the approach to the diagram editing is different. The approach discussed by Bork and Glaser is that new nodes can be added without connection to the current program. This means that a single entity relationship diagram may have multiple separated models. This behavior is not intended when using the structure-based editing approach since all states should have meaning and be ideally reachable.

## 3.6 UML Modeling Tools

One of the most common tools for software engineers is the Unified Modeling Language (UML). It is a graphical modeling language used to specify, construct, document and visualize components of software [BRJ99]. UML is under the developing of Object Management Group, which together with ISO creates the norms. UML defines most terms for modeling with a specific language and how those terms can be represented graphically. Additionally, it defines how structures and processes may be represented by the terms of a language.

### 3.6.1 Class Diagrams

A typical type of diagram is a class diagram. Class diagrams can be used to display the relationships between classes. This means that a modeler can display interfaces that are used by several classes through arrows from the interface to the class. Additionally, method names can be added to classes as well as variables. More general classes can be referenced using arrows as well. Overall this gives a good view over the relation between classes.

### 3.6.2 Visual Paradigm

Visual Paradigm International is developing the software Visual Paradigm. The diagram editor supports modeling for UML, SysML, ERD and BPMN. The software provides the possibility to create classes and interfaces for a given class diagram. The classes can be created in multiple languages including Java, C++, C# and Python<sup>5</sup>. Another feature is that a class diagram can be generated from existing code. This means that an UML Class diagram is created from a given class structure. For larger programs one can also avoid including all classes resulting in a faster creation of the class diagram. This results in the ability to perform so-called round trip engineering where one compiles the class diagram from code, if the code changes, and the code from the class diagram, if the class diagram changes, in order to keep them synchronized.

While the program provides a way to edit the class diagram and transmitting those changes to the code base the approach is still different to the KIELER project and the structure-based editing approach discussed in this thesis. The main difference is that the KIELER project forces the user to have a single main truth, which is the code. The diagram is generated from that code so it is just another representation but is not the main truth while in visual paradigm both representations may be the main truth since they need to be synchronized. Additionally, the structure-based editing approach gives the user no layout option, which is different to the freely placeable classes and interfaces in class diagrams in visual paradigm.

---

<sup>5</sup><https://www.visual-paradigm.com/features/code-engineering-tools/>





## Structure-Based Actions for SCCharts

This chapter focuses on what structure changes can be applied to SCCharts and why certain approaches were taken even if the resulting program is not correct. Some approaches omit the concept that the produced program must be correct but take a reduced requirement that the resulting program is draw able. Draw able is somewhat loose but means that within the program, once it is translated to a graph, every transition has a source and target state and that every region has an initial state. The reason why this might not be a correct program is that terminating edges in SCCharts require a hierarchical source state, which has a final state in it.

### 4.1 Successor State

One of the most common ways to add new states is to add a successor state. In the example in Figure 4.1a we want to add a successor state to the state start. This successor should only be taken if the input signal A is present. If it is present we want to set the output signal B to true and go to a new state next. The resulting program is depicted in Figure 4.1b.

The resulting model is correct if the prior model was correct. To prove that one can take a contraposition argument. Suppose the resulting model is incorrect than it must either have isolated nodes, a region must have no initial state or a state with final transition has an inner region, which has no final state. Since the added state is connected and all states where connected by correctness of the prior model the first case can not be true. The added state does not change the initial or final states so all states still exist in the resulting model and since the model was correct it must now also be correct. So we can prove the resulting model is correct if the prior model was correct.

To adjust a program using this change we need the following information from the user. First the name of the successor state. Secondly an expression that enables the transition and third an expression that sets any output signals as desired.

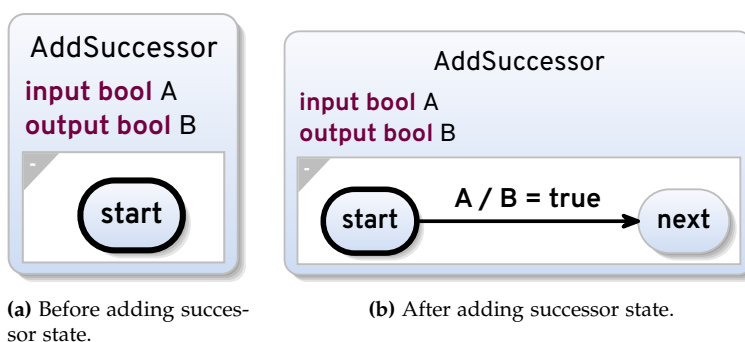


Figure 4.1. Structured change produced by add successor action.

#### 4. Structure-Based Actions for SCCharts

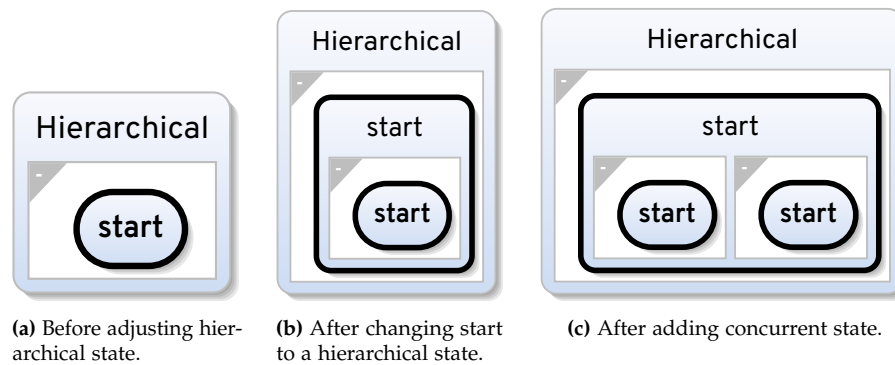


Figure 4.2. Structured change produced by add successor action.

### 4.2 Hierarchical States

Another way to add a new state is to add a hierarchical behavior to a state. This means that we introduce an inner behavior to a state. There are two ways to adjust a state in that way. First we change a state into one that has a hierarchical behavior and secondly we can introduce a concurrent region, which is executed in parallel to the already existing one.

As example for the first case we are starting in Figure 4.2a where we can change the state start so that he has an inner behavior. For this we need to add a region inside the state start and add a initial state to that region. This results in Figure 4.2c if we use the name start for the generated state. For the second case we want to introduce a parallel region to the one in that we have the state start. While also starting in Figure 4.2a we want to introduce a parallel region and the initial state inside the region should also have the name start. The resulting program is depicted in Figure 4.2b.

In both cases for the change to be made we need only two inputs from the user. First we need a name for the region, which may even be an empty string. Secondly the name for the initial state in the new generated region.

This adjustment can produce a false model. Suppose we have the model in Figure 4.3a and we want to add a concurrent region to wait for the input B. Since we have a final transition from the state waitAB each region in the state waitAB needs to have at least one final state. We could say that opening a new region in this setting will make the first state initial and final. However, this would require the user to actively reverse the change to a final state if the initial state is not supposed to be a final state. Because the initial state is most likely not the final state I decided to omit the correctness and accept a diagram, which is connected but not fully correct since the new introduced region has no final state as a result of this structural change.

While the resulting graph is not correct the graph is draw-able since every transition has an existing state as target and every region has a initial state as depicted in Figure 4.3b.

### 4.3 Initial State

To change the initial state we need to know the new state that should be initial. Additionally, we need the name of the old initial state, which we need to turn into a normal state. Changing the initial state does not void correctness of a program. However, changing of the initial state may make multiple states unreachable in a program. For example in Figure 4.3a if we make the state done in the region waitA initial we still have a correct program, but the state init is not reachable.

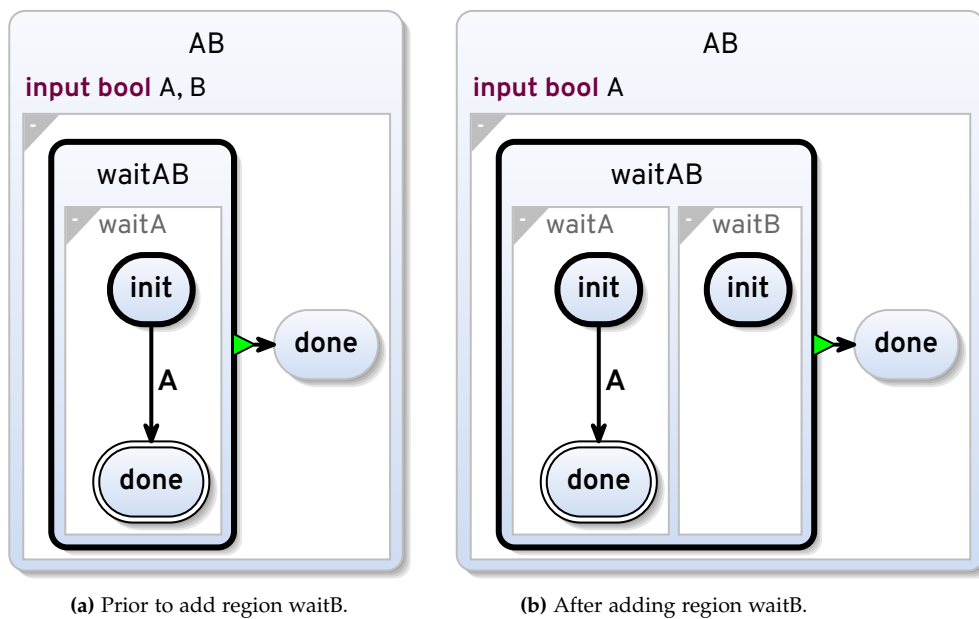


Figure 4.3. Problematic to add concurrent region waitB.

## 4.4 Toggle Final

Changing a state to a final state has a similar behavior to the making a initial state. The difference is that there may be multiple or no final state in a program. So we need only knowledge of the state that should be changed to either final or to a normal state to introduce the change.

The problem with this is that in case that there is no final state left in the region of a state and there is a final transition from that state the model may be incorrect. Suppose we have the program in Figure 4.3a and we change the state done in region waitA to a non-final state. In the resulting program the region waitA has no final state and thus since the state waitAB has a final transition to the state done the program is incorrect.

Since there is no good way to indicate what state should be final instead if the last final state is toggled. I decided to allow the change to produce an incorrect model since we can not make a good assumption on what state should be final or if that is even an wanted behavior and, therefore, do not want to add backtrack for the user for unwanted automatic changes.

## 4.5 Add Transition

Until now the structure changes would only allow a single transition between states and every state could only have a single outgoing edge. This, however, is not sufficient in most cases. Suppose we want a modulo counter witch outputs true if there was a even number of signals present and false otherwise.

In SCChart we can start in the even state as depicted in Figure 4.4a since no signals where present yet and thus we have an even amouth of signals until now. We add a successor state odd by structural change with trigger sig and output false. Now we want to add a backward edge from the state odd to the state even with the trigger being sig and as effect we set out to true. The resulting graph is depicted

#### 4. Structure-Based Actions for SCCharts

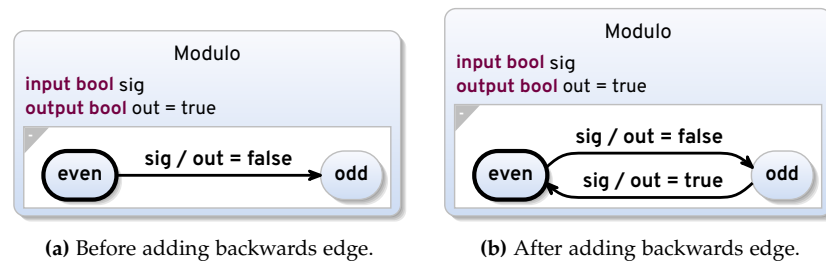


Figure 4.4. Modulo counter

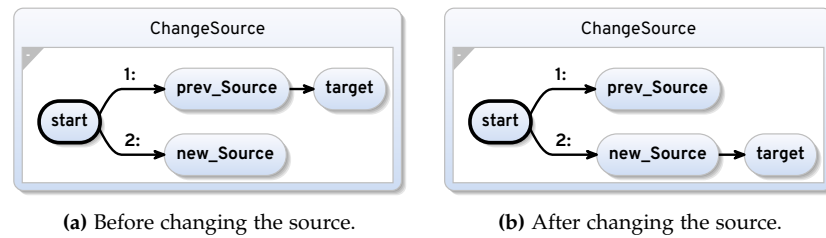


Figure 4.5. Changing the source of an transition.

in Figure 4.4b. To adjust the program accordingly we need the source and target states as well as expressions for effect and trigger.

Introducing new transitions between already existing states does not void the correctness of the model. This can be proven by assuming the resulting model is incorrect and then show that the initial model prior to adding the edges is erroneous.

### 4.6 Change Source

In contrast to the structure changes from Section 2.9 the change of source and target is separated. A simple source change is depicted in Figure 4.5. The transition from `prev_Source` to `target` is wrong and it should be a transition from `new_Source` to `target`. The structural change needs the information of the new source state as information.

This altering of transitions may introduce instantaneous loops and not reachable states both of these are not desired. Since the unreachable states may be used in later revisions and the immediate loop is highlighted as such in the IDE. The final decision is that this is ok since the immediate loop may not be executed due to transition triggers or priorities and is furthermore highlighted. Additionally, the unreachable states are not interfering with the compilation and simulation.

### 4.7 Change Target

Similar to the change source, this structure edit changes the target of a transition. A simple example is given in Figure 4.6. The transition from the initial state should go to the state `new_Target` instead of the `prev_Target`. This structure change similarly requires the `new_Targets` id.

Similar to the change source the transition change may produce instantaneous loops as well as unreachable states.

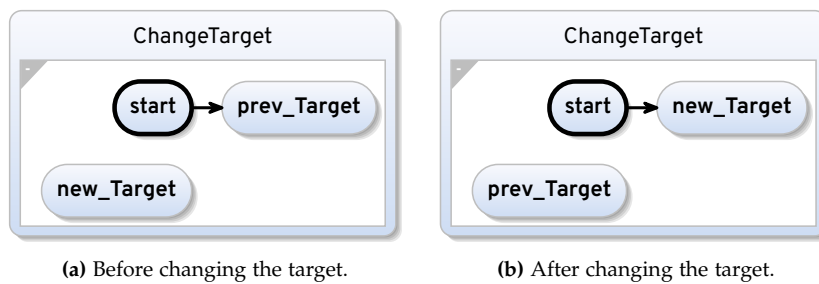


Figure 4.6. Changing the target of a transition.

## 4.8 Weak, Final and Aborting Transition

To change a transition we need to know which transition we need to adjust and to what type of transition it should be changed. Since we do not alter the source and target state the resulting graph can be drawn. The resulting program of such a change can be problematic. Suppose we have the program in Figure 4.1b and we change the transition from the state start to next to a final transition. Since the state start has no interior behavior the program is incorrect. Since there is no good way to adjust the program in those cases we allow the incorrectness.

## 4.9 Deletion

The behavior for deletion of states, transitions or regions can be problematic.

### 4.9.1 States

If we want to delete a state we need to know what state to delete. Furthermore, any edge that points to that state needs to be deleted as well. This, however, can introduce graphs where portions of the graph are separated and, therefore, some portions may be unreachable.

Another factor that needs consideration is if we delete an initial state. In that case we want to select a state in the same region and adjust it to an initial state. While we can not predict a good state to be initial we can simply select one at random. The idea here is that the extra work if we selected the wrong state as initial the extra work to change it to the desired state is the same as if the user would have changed the initial state prior to the deletion.

Deletion of a final state is a bit more problematic than the deletion of the initial state. Since we can not predict a good target to be a new final state, and we do not know if there are more final states in a region we might run into a problem. The problem is that if there is a terminating transition, and we delete the last final state we produce an erroneous program. We still allow this behavior since the resulting graph is connected and can be drawn.

The last case that is problematic is if we delete the last state from a region. Since we required that every region has a state inside it we need to delete the region from its parent state as well.

### 4.9.2 Transitions

Deletion of transitions may introduce separated portions of graphs like the deletion of states. Like for states we allow this behavior since the resulting program is still deterministic and can be drawn. In contrast to the deletion of states the deletion of transitions is independent of the transition type.

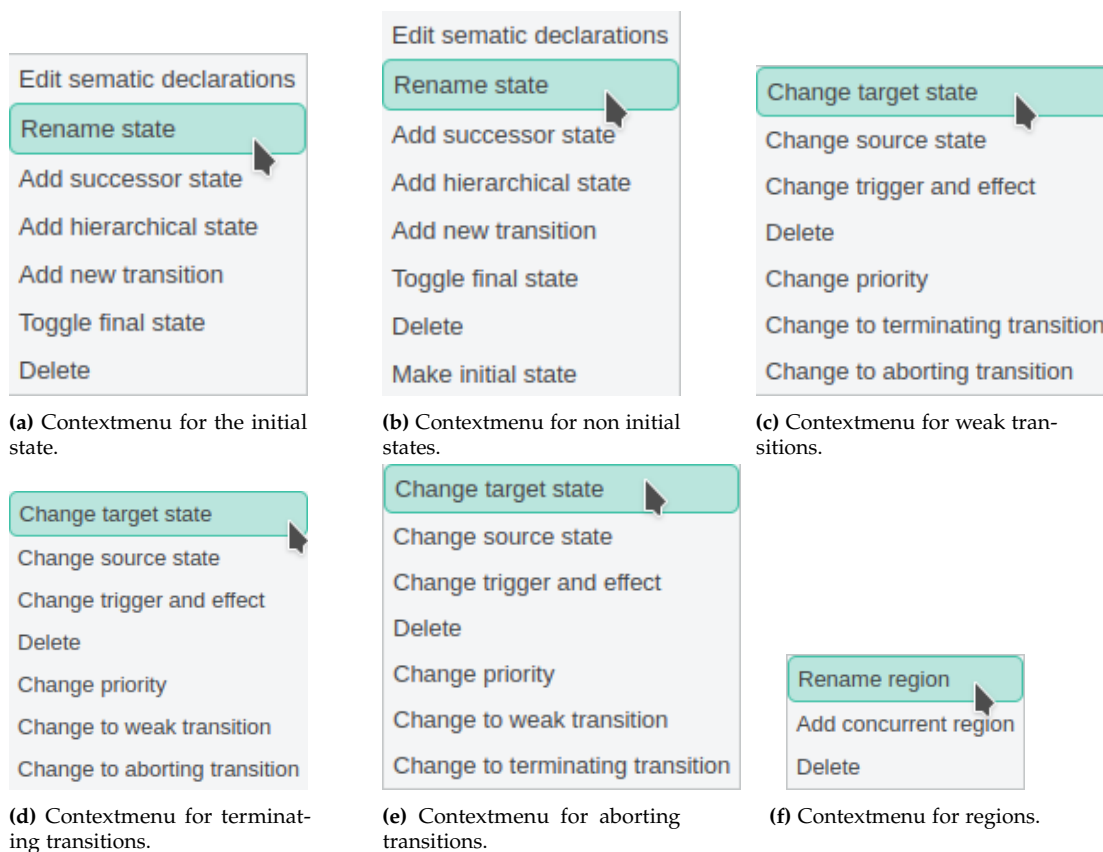
## 4. Structure-Based Actions for SCCharts

### 4.9.3 Regions

There are two cases when deleting a region: either the deleted region is one of many and in the other case it is the last region. If we delete a region from a set of regions of a state the resulting program is correct. This comes from the fact that all other regions have a correct behavior and thus it must be correct after the deletion of the region. If we delete a region it can be the last region of the state. For example if we delete the region waitA in Figure 4.3a the state waitAB is non-hierarchical. In this case we can have a problematic program since there is no interior behavior of the state waitAB, but it is required from the terminating transition. We allow this behavior since the program is still draw able and a similar problem could be generated from changing the edge type.

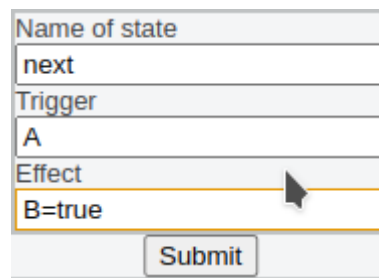
# User Story

This chapter shows the user story for every structure-based action from the previous chapter. All user interactions start by opening the context menu for a specific state, transition or region. The possible context menus are shown in Figure 5.1. The context menu is a typical feature provided in most applications, which when using the right mouse button opens a window with options. These options depend on the type of state or transition. If the state is an initial state the option to make the state initial is not shown, as depicted in Figure 5.1a. For edges different edge adjustments are shown depending on the type of edge (terminating, aborting or weak).



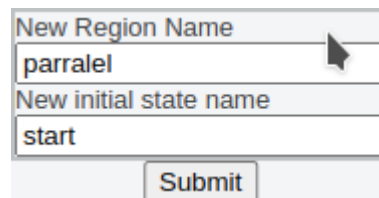
**Figure 5.1.** Contextmenus for states, transitions and regions.

## 5. User Story



Name of state
next
Trigger
A
Effect
B=true
Submit

Figure 5.2. Context menu for add successor state action.



New Region Name
parralel
New initial state name
start
Submit

Figure 5.3. Contextmenu for Add region and add concurrent region options.

### 5.1 Successor State

To do this we can open a context menu Figure 5.1a for a specific state for example the state start in Figure 4.1a and then select the Add successor state menu entry. After this a new window will open where one can name the state and set the trigger and effect as desired, which is depicted in Figure 5.2.

By submitting those changes by either using enter or clicking submit the changes then get applied to the model and the textual representation is changed furthermore the new model is displayed to the user as seen in Figure 4.1b.

### 5.2 Concurrent State

To achieve this, one can either open the context menu for the parent state, for example Hierarchical in Figure 4.2a, or open the context menu for the region itself. In the first case the option Add region will open a new window where one can enter the name for the region and the name for its initial state, as seen in Figure 5.3. In the second case the option Add concurrent region will open the same window. By Submitting the results by either hitting Enter or submitting with the button the model is changed, the text is adjusted and the new graph is displayed to the user as depicted in Figure 4.2c.

### 5.3 Add Transition

To add a backward edge we open the context menu for the state odd Figure 5.4a and select the Add new transition option from the context menu displayed in Figure 5.1c. This will add an arrow from that state to the current mouse position Figure 5.4b. We decided that the arrow should be straight and have a different color than the over edges to clearly show what transition we are currently adding. If we are dragging the edge end to the desired state even and clicking that state we can open a new window where we can insert the trigger and effect for the transition from the state odd to the state



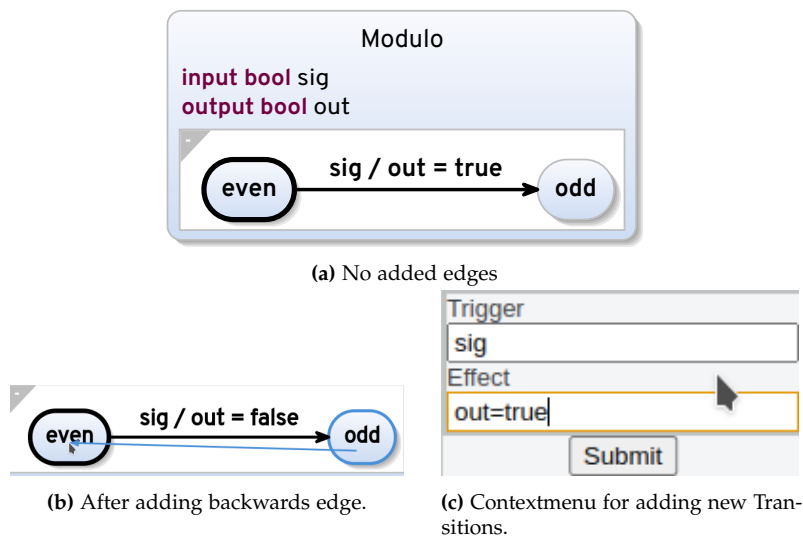


Figure 5.4. User storys for adding transitions.

even Figure 5.4c. After submitting those changes by enter or the submit button we then change the model, change the text and the new graph is displayed as in Figure 4.4b.

## 5.4 Initial State

To change the initial state one can simply select the option Make initial in the context menu, which is depicted in Figure 5.1b. This will change the previous initial state to a normal state and the selected state to be initial. For example if we open the context menu for the state new\_Start and select the option Make initial the resulting graph looks like the one depicted in Figure 5.5b. Since we force the old state to be not initial anymore the resulting graph is correct but may have unreachable states.

## 5.5 Toggle Final

Since there can be multiple final states we can display this option in every context menu for states, which are depicted in Figure 5.1b and Figure 5.1a. The information needed for the change is the state that is given by the target of the context menu. Therefore, simply selecting the option Toggle final state will change the model as desired. For example opening the context menu for the state ending and selecting the Toggle final state entry will result in the graph depicted in Figure 5.5c.

## 5.6 Change Source

To change the source of a transition the user opens the context menu for the transition(Figure 5.1c) and selects the option Change source state. Afterward, similar to the Add Transition user story an arrow is drawn from the current mouse position to the edge as depicted in Figure 5.6. If the user clicks on the desired state the structure change is performed and the model is altered. The decision to point the arrow to the edge and coming from the mouse position is preferred over the concept of having

## 5. User Story

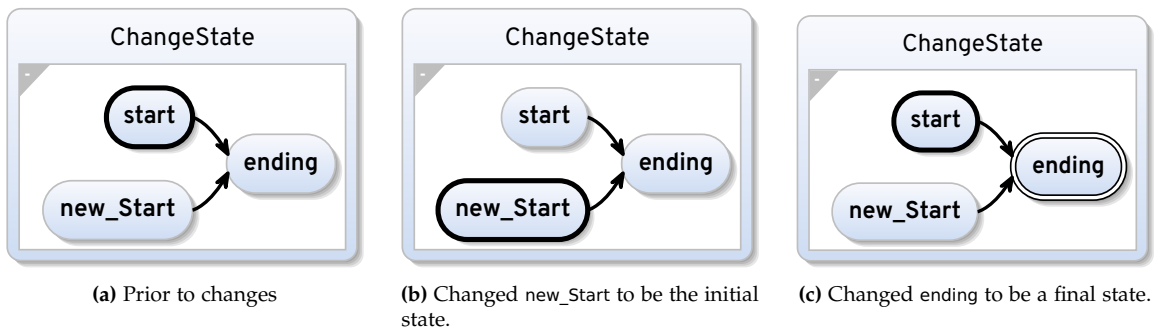


Figure 5.5. User stories for changing the state type.

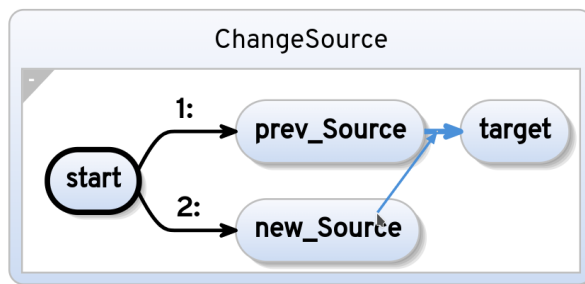


Figure 5.6. Changing the source using the arrow functionality.

the transition go to the state since it makes it more clear that the edge is altered. The behavior would otherwise be too similar to the add transition functionality and the user could get confused.

## 5.7 Change Target

The change target functionality is similar to the change source functionality the main difference is that the arrow points from the transition to the mouse position as depicted in Figure 5.7. The idea for the arrow coming from the edge instead of the state is the same as in the change source option.

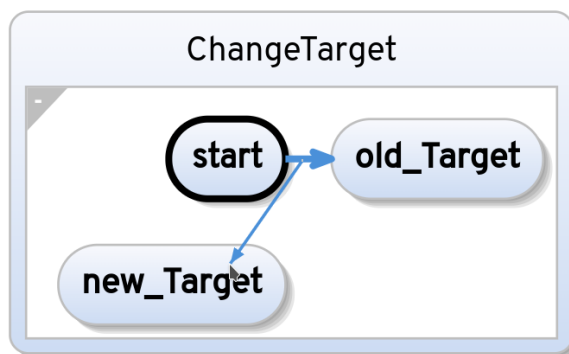


Figure 5.7. Changing the source using the arrow functionality.

## 5.8 Weak, Final and Aborting Transition

The option to change the edge only makes sense if the target type is not already the type of the edge. Therefore, we have three different context menus depending on the edge type. Since we only need the desired type and the edge id the target of the context menu and the selected type from the context menu is enough to apply the changes to the program.

## 5.9 Deletion

The deletion of a state, edge or region requires the id of the desired element. Since the id is given as the target of the context menu and the intent to delete an element is given by selecting the corresponding entry in the context menu we know everything to adjust the program and display the changes to the user.



# Implementation

This chapter explains the steps taken to implement the desired behavior into the KIELER framework. As depicted in the introduction, we need to adjust different components to achieve the desired structural editing options. The steps can be summed up as follows. Implementing an interface for classes that enable the server to fill the context menu on the client with menu entries. Since the communication between client and server is done using actions, as depicted in Section 2.3, the classes should provide the information, which actions are possible for the currently open diagram. Another step is to implement classes using the interface to provide the structure-based actions to the client. In the last step, an implementation of a SCChart specific action handler and language server extension is produced, which adjust the program according to the actions discussed in Chapter 4.

## 6.1 Client Interface

The interface to display the context menu in the VS Code extension is an addition to klighd-vscode. The idea is to enable the use of the context menu for any diagram, which can be generated if the server provides interaction possibilities.

The communication between server and client is displayed in Figure 6.1. In the first step, a client request a model to be generated. After the generation of the diagram a number of StructuredEditMsgs can be appended to the root of the model. The StructuredEditMsgs define what menu entries are displayed in the context menu for the diagram type. To achieve that, the StructuredEditMsgs define what inputs are needed from the user as well as the variable name in that the input should be stored. Additionally,

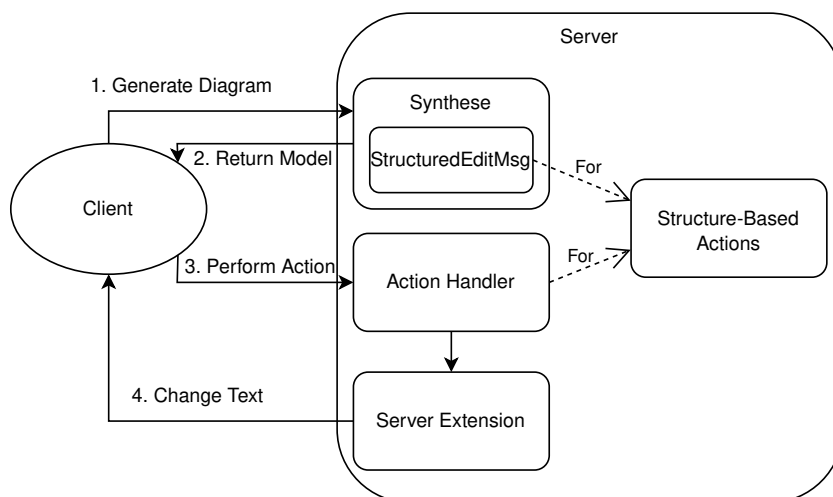


Figure 6.1. Steps performed for structure-based editing.

## 6. Implementation

```
1 public class StructuredEditMsg {
2
3     String label;
4     String kind;
5     Boolean mergable;
6     InputType[] inputs;
7
8     public StructuredEditMsg(String label, String kind, Boolean mergable,
9         InputType[] inputs) {
10        this.label = label;
11        this.kind = kind;
12        this.mergable = mergable;
13        this.inputs = inputs;
14    }
15 }
```

**Listing 6.1.** A set of this class is assigned to each state, edge and region.

```
1 map.put(SCChartsSemanticFilterTags.STATE, #[
2     EditSemanticDeclarationAction.getMsg(),
3     RenameStateAction.getMsg(),
4     AddSuccessorStateAction.getMsg(),
5     AddHierarchicalStateAction.getMsg(),
6     AddTransitionAction.getMsg(),
7     ToggleFinalStateAction.getMsg(),
8     DeleteAction.getMsg()
9 ])
```

**Listing 6.2.** Adding all StructuredEditMsgs for the type state to the map.

the StructredEditMsg's contain the unique KIND of the actions. In this way each StructuredEditMsg can be used to create an action containing the inputs from the user. By sending such an action to the server, which refers to the third step in Figure 6.1, the server can handle the action according to the action handler and server extension and perform updates.

The structure of the StructuredEditMsg class, which is defined on the server and send to the client, is depicted in Listing 6.1. The `label` refers to the string displayed in the menu entry. The `kind` refers to a unique string, which is used to differentiate between actions. While the `inputs` array defines what inputs are taken from the user. Each StructuredEditMsg is assigned to a specific type of graphical element. In that way the server chooses what entries should be shown if the context menu is opened for different graphical elements. To achieve this tags, which where discussed in Section 2.5, are used. Specifically, all StructuredEditMsgs that are assigned to a specific tag are put into an array and added to a map with the tag as its key. For example all StructuredEditMsgs that are possible for states are added to the map with the key `SCChartsSemanticFilterTags.STATE`, as depicted in Listing 6.2.

The InputType class is depicted in Listing 6.3. The client currently supports three different input types. First selecting a target node, second selecting a source node, and third simple string inputs. By setting the `typeOfInput` to `String`, `SelectSource`, or `SelectTarget`, the application knows what input type to perform. The result of the input operation of the user is stored in a new variable, which is named after

```

1  public class InputType {
2      String field;
3      String typeOfInput;
4      String label;
5
6      public InputType(String field, String typeOfInput, String label) {
7          this.field = field;
8          this.typeOfInput = typeOfInput;
9          this.label = label;
10     }
11 }

```

**Listing 6.3.** The class is used to tell the client what inputs are needed.

the string stored in the `field` variable. Finally, the `label` is used when a string input is requested. In this case the content of the `label` variable is displayed to let the user know which string is needed. An example for the add transition action is given in Figure 6.2. Line two results in a selection process displayed in Figure 6.2a. Line three and four result in the string input interface depicted in Figure 6.2b. After using the submit button or hitting the enter key, the client creates an action containing the inputs from the user. In the case of the inputs in Figure 6.2, it results in the action depicted in Listing 6.5.

To achieve the interface two new components are added to `klighd-vscode`. The `ContextMenuProvider` is responsible for drawing the context menu and handling interactions with it. The `graphprogrammingMoveMouseListener` is enabled by the context menu when a select source or target is requested by a `StructuredEditMsg` and an arrow should be drawn in the diagram. Both components use DOM manipulation to draw information in the view.

### 6.1.1 ContextMenuProvider

Once the user opens the context menu the DOM is searched for the context menu id. If there is a context menu there is no need to create a new context menu. If no context menu element is in the DOM we create one and use DOM manipulation to add it to the diagram element with the id `keith-diagram_protty`. This ensures that the context menu is only displayed while using the `klighd-vscode` extension.

Since the context menu is invoked for a specific element in the graph, we already get the element id. Through the use of the selected elements tag the type of element can be extracted. Using the information of the selected type, the sets of `StructuredEditMsgs` are filtered for those that refer to the specific element. For every `StructuredEditMsg` that is assigned to the tag of element a new entry is added to the context menu. For example the seven `StructuredEditMsgs` displayed in Listing 6.6 are assigned to the tag `STATE`. Therefore, entries with the seven labels are added to the context menu.

Upon selecting a context menu entry, the context menu content is deleted and refilled with the corresponding `InputTypes`. For all string inputs the label and an input field are added to the context menu. For selection input types the `graphprogrammingMoveMouseListener` is enabled to draw the graph and the context menu is hidden until target or source is selected. After the `graphprogrammingMoveMouseListener` has returned a target the remaining context menu entries are displayed to the user. For example, in case of the add new transition action, after selecting the target the user needs to input a trigger and effect as displayed in Section 5.3.

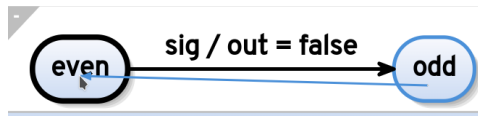
## 6. Implementation

```

1 def static InputType[] getInputs(){
2     val input1 = new InputType("destination", "SelectTarget", "Destination");
3     val input2 = new InputType("trigger", "String", "Trigger");
4     val input3 = new InputType("effect", "String", "Effect");
5     return #[input1, input2, input3];
6 }

```

Listing (6.4) Example for input types for the AddTransition Option.



(a) Selection of a target for a new transition.

Form (b) shows a web interface for adding a transition. It has two input fields: 'Trigger' with the value 'sig' and 'Effect' with the value 'out=true'. Below the fields is a 'Submit' button.

(b) String inputs for trigger and effect for the new transition.

```

1 NewServerActionMsg{
2     KIND = 'SCChart_graph_AddTransition'
3     destination = 'id of even'
4     trigger = 'sig'
5     effect = 'out=true'
6 }

```

Listing (6.5) Resulting action

Figure 6.2. Definition of inputs for the Add Transition action.

```

1 0: {label: 'Edit semantic declarations', kind: 'SCChart_EditSemanticDeclarations', mergable:
   false, inputs: Array(0)}
2 1: {label: 'Rename state', kind: 'SCChart_graph_RenameState', mergable: false, inputs: Array
   (1)}
3 2: {label: 'Add successor state', kind: 'SCChart_graph_AddSuccessorState', mergable: false,
   inputs: Array(3)}
4 3: {label: 'Add region', kind: 'SCChart_graph_AddHierarchicalState', mergable: false, inputs:
   Array(2)}
5 4: {label: 'Add new transition', kind: 'SCChart_graph_AddTransition', mergable: false, inputs:
   Array(3)}
6 5: {label: 'Toggle final state', kind: 'SCChart_graph_MakeFinalState', mergable: false,
   inputs: Array(0)}
7 6: {label: 'Delete', kind: 'SCChart_graph_Delete', mergable: true, inputs: Array(0)}

```

Listing 6.6. StructuredEditMsgs for the tag STATE.



```

1 export interface NewServerActionMsg extends Action {
2     kind: typeof NewServerActionMsg.KIND;
3     [key: string]: any;
4 }

```

**Listing 6.7.** Action interface to send any action to the server.

### 6.1.2 Arrow Drawing

The arrow drawing is done using DOM manipulation as well. In order to display an arrow to the current mouse position a SVG element is added to the SVG in that the diagram is displayed. I decided to make the arrow straight instead of having it routed by the server since the straight arrow has the benefit that it has a unique appearance in the graph. After the selection of an element the target id is sent to the *ContextMenuProvider*. After the *ContextMenuProvider* received all inputs a new action is created.

### 6.1.3 Action Creation

The collected information from the user, which is defined by the inputs array, needs to be added in a new action. The problem with creating an action during runtime is that the action kind needs to be assigned to a specific action handler. Since the action kind is unknown prior to receiving a model, the actions can not be registered at any action handler. Another problem is that we need to assign values to fields on actions during runtime, which may not exist since the fields are also only known upon receiving the input array.

The first issue is solved by using the server action handler directly instead of calling the action dispatcher. However, this introduces unintended behavior with regard to Sprotty since the viewer should only access the action dispatcher to dispatch any actions created in the viewer, showing as the dashed line in Figure 2.4. Since the actions can not be registered prior to receiving the graph from the server this proves infeasible and the direct access to the server action handler is implemented. The second issue is resolved using a simple interface, as depicted in Listing 6.7. The map allows the user to define any new fields during runtime.

## 6.2 Server Side

As depicted in Figure 6.3, we have dependencies from the language server to the SCChart-specific synthesis. This means that importing the implementations for language specific functionality results in a dependency loop. Since the implementations are necessary for the language server, service loaders are implemented. A service loader is used to load classes that implement a specific interface during runtime. Implementing a service loader for *IActionHandler*, *IStructuredActions* and *IStructuredProgrammingLanguageServerContribution* allows the use of language specific action handlers, language server extensions and actions. This allows to register actions for receiving in the *KGraphTypeAdapterUtil* and also to forward those actions to the correct action handler in the *KGraphDiagramServer*. Finally, new language server extensions can be loaded in the *LSCreator* to introduce extensions handling changes to the model or communication with the client. The *LSCreator* is responsible for the creation of a new language server. The *KGraphDiagramServer* is the distributor of actions to the corresponding action

## 6. Implementation

handlers and the `KGraphTypeAdapterUtil` is used to register new actions so they can be handled by the `KGraphDiagramServer`.

Implementation of new languages for structure-based editing can be achieved by programming the language specific features in the corresponding synthesis. Afterwards, the new implementations can be added to the service loader.

Since the actions are already in the language-specific synthesis on the server, we can use a *SynthesisHock* to add the map with the tags and `StructuredEditMsgs`, discussed in Section 6.1, to the root of the model. For the tag state this is displayed in Listing 6.2. We only append the map to the root since this reduces the data that needs to be transmitted to the client since for all other elements of the graph we only need to transmit the tags, which are assigned to the corresponding element.

### 6.2.1 Server Action Handler

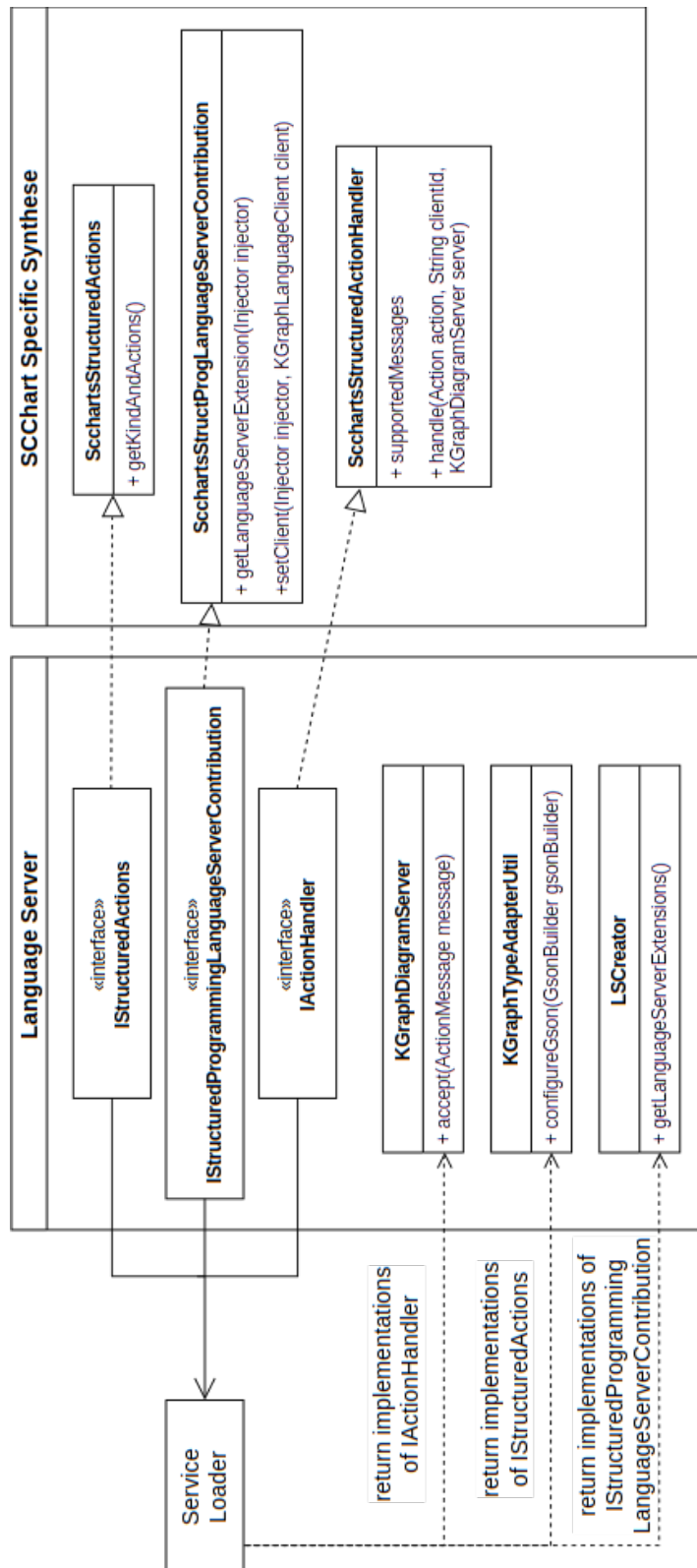
The actions are received at the `KGraphDiagramServer` and need to be handled by an action handler. Therefore, a `ScchartsStructuredActionHandler` is implemented. This action handler handles the `SCCharts`-specific structure-based action messages. The `ScchartsStructuredActionHandler` forwards the actions to the specific methods provided by the `ScchartsStructuredProgrammingServerExtension`.

### 6.2.2 Structural Editing Server Extension

The `StructuredProgScchartLanguageServerExtension` provides the functionality to change the textual representation given a specific action. The received action from the client is always containing the id of the selected element. The id is unique and can be used to get the *KNode*, in case of states and regions, or the *KEdge*, in case of transitions. The *KGraph* is the graph that is transmitted to the client. Altering it would only update the view while not updating the textual representation. In order to update the textual model, we need to alter the domain model.

The domain model is build in a tree like structure for states and regions. Regions contain states and states contain regions. The transitions are registered at the source and target states. This can be used to alter the domain model according to the structure-based actions given in Chapter 4. To achieve this, new states, regions, and transitions can be created using already existing factory implementations for `SCCharts`. One of the main problems, however, is the creation of new triggers and effects for transitions. Triggers and effects operate on valued object references. While creating a new trigger and effect is possible the reference is pointing to a dummy. Therefore, every trigger and effect needs to be updated so that the references point at the correct valued object.

After adjusting the domain model according to a given action, it can be used to generate a textual representation of the new model. By sending the updated textual representation to the client the old content is updated and the structure change is performed.



**Figure 6.3.** Implemented interfaces and corresponding SCChart-specific components. Other languages can be added by updating the service loader and implementing similar classes to the components on the right.



# Evaluation

This chapter explores the usability of the structural editing software. To achieve an objective answer the clicks and inputs of a user of each structure-based action is compared to the textual interactions needed for changing the model in the textual representation. Furthermore, a group of inexperienced programmers in the field of SCCharts is analyzed for two different test programs. The first half of the group is asked to program using the structure-based editing software while the over half is asked to program using the textual representation. The two groups are then analyzed for understanding the program and how fast the program is created. In the last step of the evaluation I compare the actions needed for structure-based editing to the actions when using common graph editors where placing new elements is done directly in the graph.

## 7.1 Clicks for Structure Changes

The structure-based programming approach should have the least number of context menu actions to perform a specific structure change. The inputs required for a structure-based action are composed of first opening the context menu followed by selecting a specific action from the context menu. Afterwards, the inputs for the action are requested from the user. The table Table 7.1 depicts the inputs needed for the structure-based actions depending on doing them in the text or using the structure-based editing approach. In the table the context menu input refers to opening and selection of a action from the context menu. The equal column depicts what mouse or keyboard inputs differ in the approaches to make a certain structure-based action.

Comparing the two input requirements for the editing approaches we can see that in most cases the inputs for the keywords are compared with the context menu action. Since the context menu action should be overall faster since the keywords need more inputs it could be expected that the structure-based approach is faster. This, however, is not the case since auto-completion for keywords as well as state names can make up for the difference. Additionally, in some cases one can use copy and past to copy almost identical transitions. Overall the inputs for context menu can not be reduced anymore since the context menu opening takes the first action and the second input for selecting a structure-based action is needed as well. All over inputs are also required for the textual editing approach, which means they are necessary as well.

## 7.2 Test

The testing for the usability and ease of use is conducted over a group of teen people. These people have a background from computer science but no prior knowledge of SCCharts. The group is separated in two groups of five where one test group is using the structure-based programming approach and the other group is editing the textual model directly.

Both groups are given an introduction into SCCharts inform of the ABRO example, which is explained

## 7. Evaluation

	Text editing	Structured editing	Equal
Rename	deleting state name + string input for state name	context menu + string input for state name	deleting state name = context menu
Successor State	input if trigger do effect go to state name + finding place for state + input state + state name	context menu + string input for state, trigger, effect	input if do go to + state = contextmenu
Concurrent State	input region + region name + input initial state and state name	context menu + string input for region and state name	input region + initial state = context menu
Add Transition	input if trigger do effect go to state name	context menu + selecting target + string input trigger and effect	input if do go to + state name = context menu + selecting target
Initial State	delete initial from old state + input initial for desired state	context menu	delete initial from old state + input initial for desired state = context menu
Toggle Final	input final or delete final	context menu	input final or delete final = contextmenu
Change Source or Target	delete old state name + input state name	context menu + selecting new target or source	delete old state name + input state name = context menu + selecting new target or source
Change to weak transition	delete old keyword	context menu	delete old keyword = context menu
Change to aborting or terminating transition	delete old key word + input new keyword	context menu	delete old key word + input new keyword = context menu
Deletion States	delete state + delete all edges	context menu	delete state + delete all edges = context menu
Delition Edges or Regions	delete edge or region	context menu	delete edge or region = context menu

**Table 7.1.** Table with number of inputs from user depending on the editing approach and the structure change.

in the next section. After the explanation the test groups are given two programs to build. The first program should give an easy entry and is a simple modulo signal giver explained in Section 7.2.2. The second is an elevator program that controls the direction of an elevator and is explained in Section 7.2.3. During the construction of the programs the users can ask any number of questions regarding the structure of SCCharts for graphs or text.

Monitoring the number of questions as well as the time needed for the construction of the program gives an indicator to how useful the software is for learning the programming language SCCharts. Additionally, the users are expected to explain the created software while using graph and text. Furthermore, the users can rate the usability of the software.

### 7.2.1 ABRO

ABRO is one of the first programs that compose of most of the features SCCharts provide and is, therefore, suited to explain all concepts. The behavior of ABRO is that after an initial tick the program will wait for the inputs A and B, which may happen at separate ticks and output O if both signals have been present. If during execution the signal R is present the program resets, which means if one input was present already it needs to be present again. After the first emitting of O the program waits until R is present to reset the program and wait for the inputs A and B again.

Both explanations of ABRO follow the top-down approach for programming. This means that first a state ABO is created. Afterwards, the aborting self transition of ABO with the trigger R. Then the state ABO is elevated to a hierarchical state with the initial state waitAB and a transition with output O to a second state done is implemented. The state waitAB is then elevated to the hierarchical state with two regions waiting for the inputs A and B separately. The main difference in the explanation is that for one group the structure-based editing approach is taken during building the ABRO application while for the other group the explanation is given for the graph while implementation is done textually in the code.

### 7.2.2 Modulo

The modulo example can give a good first start because of its simplicity. The idea of the modulo signal giver is that the program should send the output signal  $O=true$  if the number of previous input signals S until now is even and  $O=false$  if the number of signals is odd. Additionally, the program should be reset-able with the input signal R. The given start program for both groups is given in Listing 7.2.

This program can be realized using only three states, which makes it rather simple. An example program is given in Figure 7.2. The idea of the example program is to distinguish between the two states odd and even. The odd state means that an odd number of signals was true until now while the even state means that the number of signals is even. With those two states we can set the output signal O to the desired value. The reset is achieved using the hierarchical state modulo with the aborting self transition similar to the ABRO example.

The value behind implementing this smaller program is to get a first feeling for SCCharts and the idea behind the state based programming. While the programs are small this also gives an indication for the ability to understand the principle of SCCharts depending on the use of structure-based programming and traditional textual editing.

### 7.2.3 Elevator

An elevator is a more advanced program since there are more inputs more outputs and more states to consider. The behavior of the elevator is as follows. The elevator should move in the direction of the

## 7. Evaluation

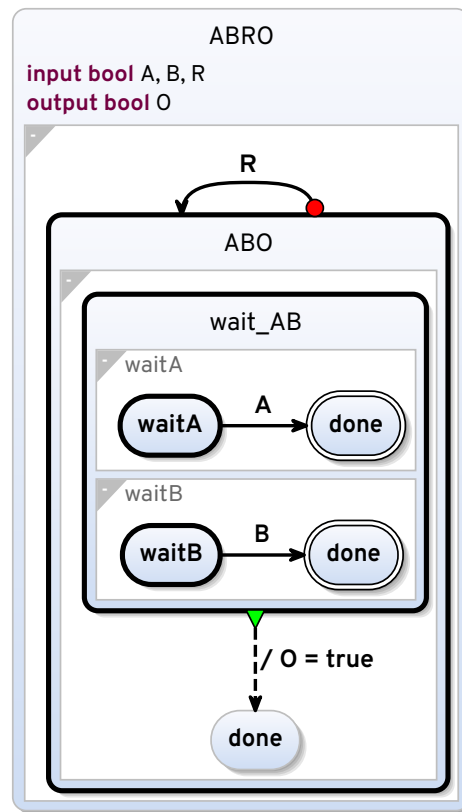


Figure 7.1. ABRO as graph

clicked button (floor one to three). To do so the current position is given as an integer signal and the direction is supposed to be written in an integer output. The value of the output is deciding factor for the direction (minus one is down, zero is stationary and one is up). The desired location is also given as an integer input. If a user is clicking the emergency button the elevator should stop for one tick and then return to the direction it was moving. The given starting point for both groups programs is depicted in Listing 7.4.

To realize the elevator one can use four states as depicted in Figure 7.3. The first three states distinguish between the directions in that the elevator may move. This means we have states for up, down and standing still. In case of a new `desired_floor` input we can decide on the new direction by comparing the current floor and the wanted floor. The last state is an emergency state. Triggering the emergency button in any state results in going to the emergency state and stopping the elevator by setting the direction to zero. Since the desired floor and current floor are still known after the emergency stop we can go back in the direction that is correct.

The idea behind this test is that it should indicate if the users can oversee a larger number of states depending on the usage of structure-based programming. This also indicates if structure-based programming can help writing complexer programs with less time effort in learning. The resulting programs explanation should also give insight into the understanding of text and graph of both groups.



```

1 scchart ABRO {
2   input bool A,B,R
3   output bool O
4
5   initial state ABO {
6     initial state wait_AB {
7       region waitA {
8         initial state waitA
9         if A go to done
10        final state done
11       }
12      region waitB {
13        initial state waitB
14        if B go to done
15        final state done
16      }
17    }
18    do O=true join to done
19    state done
20  }
21  if R abort to ABO
22 }

```

Listing 7.1. ABRO as textual model.

```

1 scchart ModuloSig{
2   input bool S,R
3   output bool O
4 }

```

Listing 7.2. Given code for modulo counter.

## 7.3 Test Results

While both groups were asking similar questions in the beginning regarding the behavior of the transition types and the behavior of transitions the group using textual editing needed to ask more questions regarding the syntax. Since most part of the syntax is abstracted by the structure-based editing approach the other group did not have any problems editing the program.

While the group using the text editing approach asked on average seven questions more than the group using structural editing the difference comes mainly from the questions regarding syntax, which can be seen in Table 7.2. This makes sense since the structure-based editing approach abstracts from the syntax. This also proves the idea that syntax is one of the key problems when starting to learn new concepts [Tsa18].

This is also visible when comparing the average time necessary for both groups to program the test examples in the last column of Table 7.2. The first example took the textual editing group on average two minutes more and the second test program took 2.8min more. These time differences came mostly from the questions regarding the syntax.

## 7. Evaluation

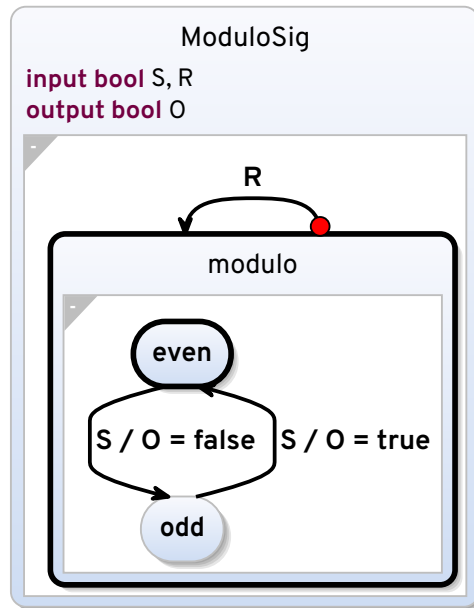


Figure 7.2. Example graph for modulo test.

```
1 scchart ModuloSig{
2   input bool S,R
3   output bool O
4
5   initial state modulo{
6     initial state even
7     if S do O = false go to odd
8
9     state odd
10    if S do O = true go to even
11  }
12  if R abort to modulo
13 }
```

Listing 7.3. Example code for modulo counter.

```
1 scchart Elevator{
2   input bool emergency
3   input int desired_floor,current_floor
4   output int direction
5 }
```

Listing 7.4. Given code for elevator.

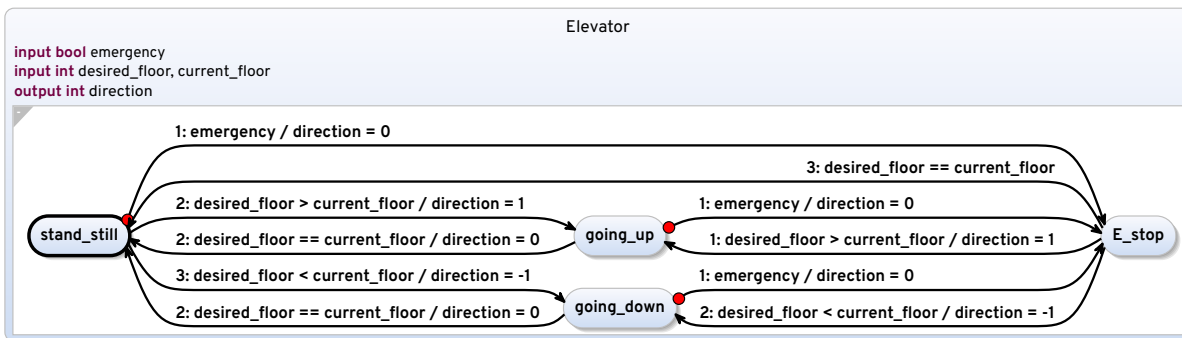


Figure 7.3. Elevator as graph

```

1  scchart Elevator{
2    input bool emergency
3    input int desired_floor,current_floor
4    output int direction
5
6    initial state stand_still
7    if emergency do direction = 0 abort to E_stop
8    if desired_floor > current_floor do direction = 1 go to going_up
9    if desired_floor < current_floor do direction = -1 go to going_down
10
11
12   state going_up
13   if emergency do direction = 0 abort to E_stop
14   if desired_floor == current_floor do direction = 0 go to stand_still
15
16   state going_down
17   if emergency do direction = 0 abort to E_stop
18   if desired_floor == current_floor do direction = 0 go to stand_still
19
20   state E_stop
21   if desired_floor > current_floor do direction = 1 go to going_up
22   if desired_floor < current_floor do direction = -1 go to going_down
23   if desired_floor == current_floor go to stand_still
24 }

```

Listing 7.5. Example solution for the elevator.

## 7. Evaluation

	average questions	average questions to syntax	average questions to behavior of sccharts or program	average time needed for programming in min
Text editing	25	7	18	7 + 16.4
Structural editing	18	1	17	5 + 13.2

Table 7.2. Table with the amount of questions for both tests.

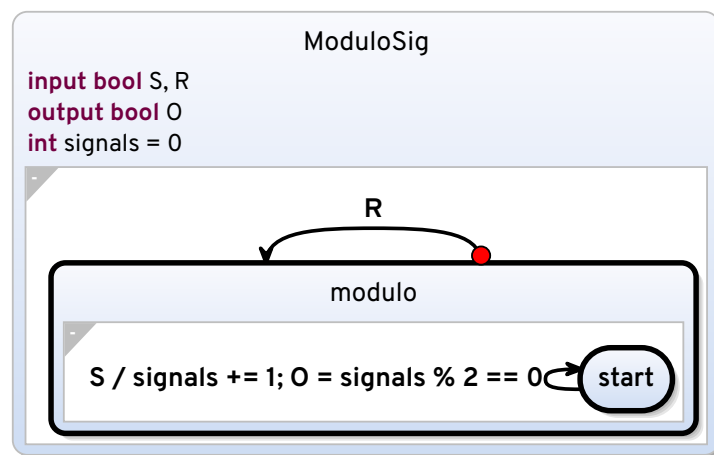


Figure 7.4. Graph solution for modulo test from text editing user.

Another factor, which is observable from comparing the text editing and the structure-based editing approach of the two groups, is that the text editing group is not focusing on the state machine idea embedded in SCCharts when compared to the structure-based editing group. This results in single state ideas as depicted in Figure 7.4. From the textual editing side this approach makes sense since the solution has fewer lines and is still clear when seeing the code. However, the single state diagram is less clear when compared to the example solution. This kind of programming was not observable from the group using structure-based editing.

This overall indicates that the structure-based programming approach is simpler to understand in the beginning because fewer questions were needed, and less time was spent programming the test cases. Therefore, the structure-based programming approach may improve the learning curve for students learning the concept of SCCharts. Additionally, important concepts like state machines may be simpler to focus on.

While the structure-based editing approach may seem simple it might be good to compare the graph creation using structure-based editing to creation using traditional graph editing programs.

## 7.4. Comparing Structure-Based editing to Traditional Graph editing

```
1 scchart ModuloSig{
2   input bool S,R
3   output bool O
4   int signals = 0
5   initial state modulo {
6     initial state start
7     if S do signals += 1; O = (signals%2)==0 go to start
8   }
9   if R abort to modulo
10 }
```

Listing 7.6. Code solution for modulo test from text editing user.

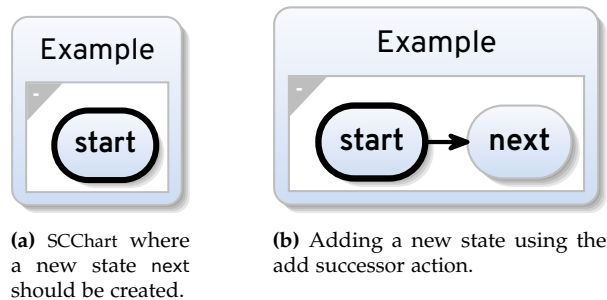


Figure 7.5. Automatic resizing and placement of nodes in a place efficient way.

## 7.4 Comparing Structure-Based editing to Traditional Graph editing

The traditional graph editing approach requires the user to place and resize nodes and perform the layout for the edges. An example software for traditional graph editing is diagrams.net, which is depicted in Figure 7.6. The idea is that one can select any shape from the side panel and place it anywhere in the graph. This can produce any graph that only includes the shapes which are provided. The user, for example, can produce a similar diagram to the one depicted in Figure 7.5a. The problem with this is that editing the graph is tedious and time consuming.

For the example depicted in Figure 7.5a creating the next state only requires the inputs for state name i.e. next and the input for the context menu and results in the graph depicted in Figure 7.5b. When simulating the same transition in diagrams.net, the placement of a new node inside the Example shape requires reshaping the Example shape to make room for the new state and the creation of two new shapes. One shape is the box and the text and the other shape adds the edge which goes from the shape start to the newly created one. Additionally, the texts need to be placed uniquely as well.

This means that an add successor action requires reshaping of the parent state, creation of three new elements and reshaping of them in order to be place efficient. There are even more problems when programming with a more hierarchical architecture since reshaping one shape at the bottom, so it takes more place, may require to reshape the parent and afterwards the reshaping of its parent. The possibility of using copy and paste are limited as well, since, for example, the shape start inside the shape region is smaller since it is on another hierarchical layer.

Overall this shows the simplicity of producing high quality graphs using the structure-based editing approach.

## 7. Evaluation

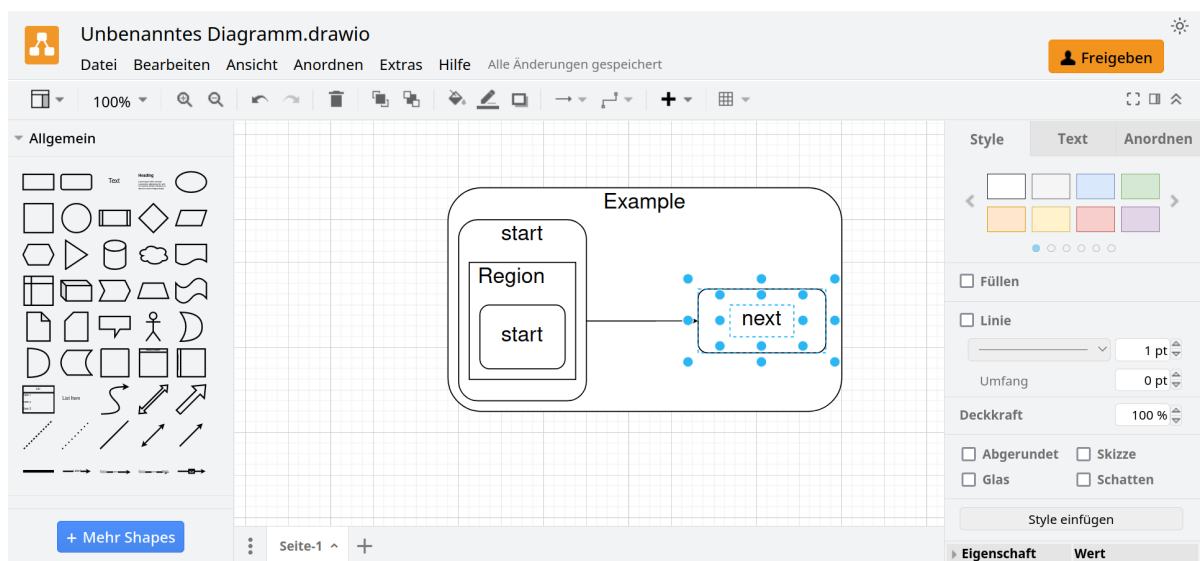


Figure 7.6. diagrams.net application with the text and shape on the right selected.

# Conclusion

This thesis shows a way to utilize the structure-based editing approach for SCCharts. The focus of this thesis is the user interaction with a diagram using the context menu in order to adjust the meaning of the program behind the displayed diagram. In this chapter the structure-based actions the user interaction and the test results are summarized. Additionally, still existing problems as well as transporting the idea of structure-based editing to new languages is presented in future work.

## 8.1 Summary

The user can interact with the diagram view using a context menu. To achieve this a UI is added, which can display StructuredEditMsgs. A StructuredEditMsgs contains the inputs necessary to produce a specific structure change. Depending on the selected StructuredEditMsgs from the context menu more user inputs are taken. This may be by selecting a new node using another UI or by simple string inputs. The communication to the server is realized using a unique action containing the inputs from the user as well as the kind specified by the StructuredEditMsgs. Upon receiving an action from the client a StructuredActionHandler takes the action and forwards it to a StructuredServerExtension. The server extension then updates the model depending on the received action. If the inputs are faulty a message is sent to the client with the specific problem of the input. Finally, the server extension produces a new textual representation of the updated model and sends it to the client to update the contents of the file. Evaluating the implemented structure-based editing approach shows that the concept is user-friendly and helpful when starting to learn SCCharts. Additionally, the main concept of state machine type programming may be more prominent when using the structure-based editing approach.

## 8.2 Future Work

One obvious future addition is implementing the idea for structure-based editing for other languages. This would require deciding what structure-based actions are possible. Adding the StructuredEditMsgs to the root of the model for that specific language. Implementing an action handler and a server extension for that specific language. And finally adding actions, action handler and server extension to the service loaders. As one can see there is no need to adjust the client since the interface provided by the client allows the server to fill it with new information.

Since the context menu interface allows the creation of any action during runtime the context menu could be used for different additions. I want to present two ideas for using the context menu during the simulation but the context menu could also be used in other ideas as well.

During simulation the current setup is depicted in Figure 8.1. On the left, in the KIELER Simulation tree, one can set the input signal to the program for the next tick. This allows the user to simulate a specific sequence of inputs to a specific program. The current states are depicted with a red border on the right hand side. While this allows the user to run through the program in a specific way it forces

## 8. Conclusion

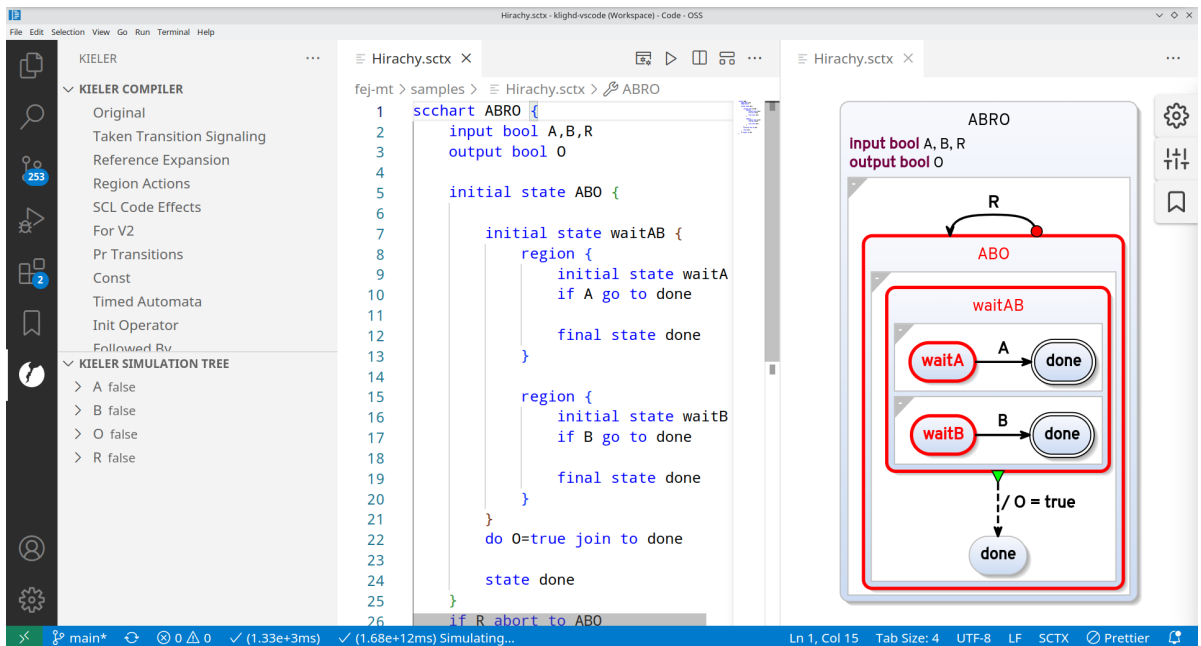


Figure 8.1. Simulating a SCChart

switching between graph and input panel. The idea I propose is that the user can open the context menu for a specific edge from the current state in the simulation and select that this edge should be taken, as depicted in Figure 8.2a. The server enables an input that forces taking that transition and a new state is reached.

The interactive simulation could yield overall faster steps since the inputs could be set directly from an transition without thinking about the actual input parameters. Additionally, the server could give insight into all inputs that would enable the transition instead of only setting one specific input.

The second idea is not fully build into the simulation but rather extracts all possible input sequences that would end in a specific state. To achieve this behavior a menu entry Show possible paths could be added to the context menu, as depicted in Figure 8.3. Once this entry is activated the server uses a model checking algorithm to identify all sequences that end in the state done. This would allow the user to identify bugs where after a certain input sequence a state is reached in an unwanted fashion.



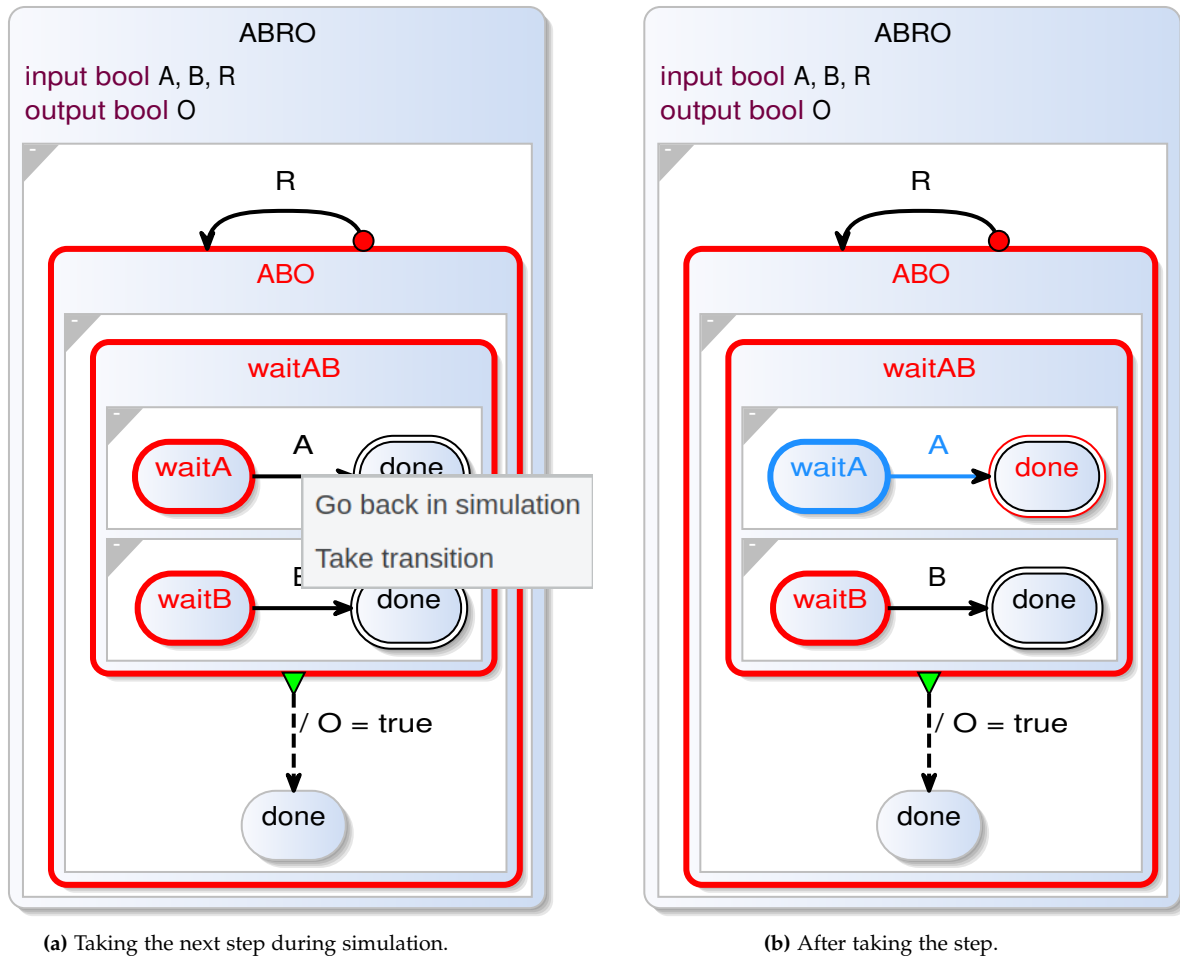


Figure 8.2. Interactive simulation

## 8. Conclusion

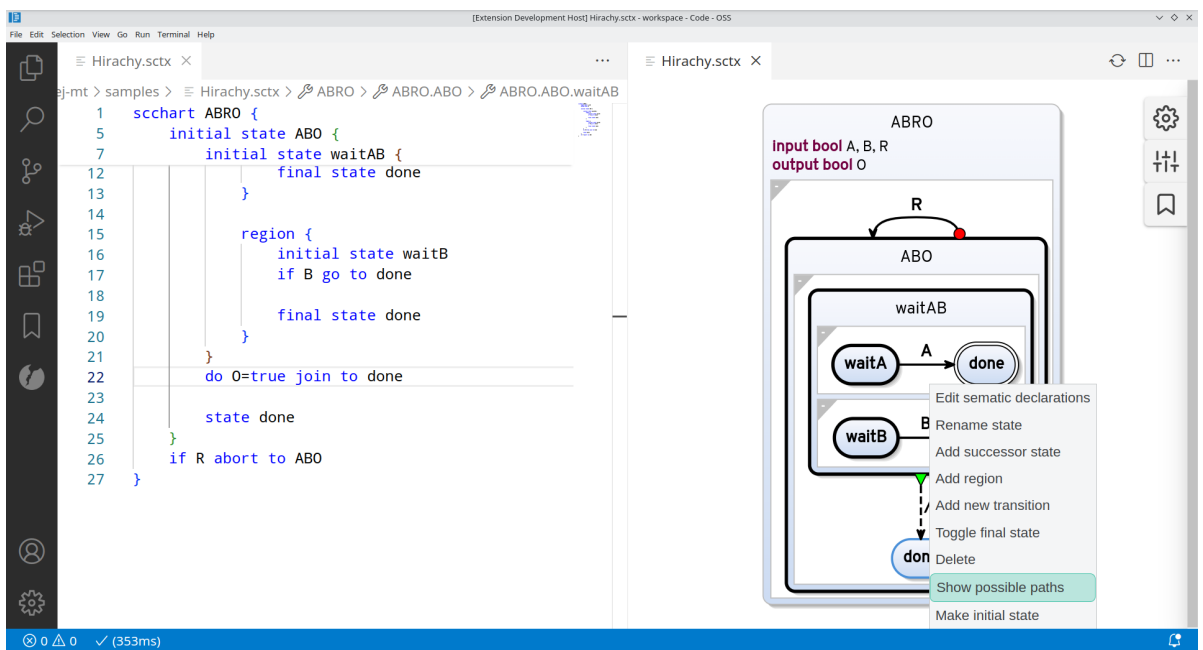


Figure 8.3. Obtaining all possible input paths to state done.

# Bibliography

- [ACL+17] Lorenzo Addazi, Federico Ciccozzi, Philip Langer, and Ernesto Posse. “Towards seamless hybrid graphical–textual modelling for uml and profiles”. In: *Modelling Foundations and Applications*. Ed. by Anthony Anjorin and Huáscar Espinoza. Cham: Springer International Publishing, 2017, pp. 20–33. ISBN: 978-3-319-61482-3.
- [BRJ99] Grady Booch, James Rumbaugh, and Ivar Jacobson. “Unified modeling language user guide, the (2nd edition) (addison-wesley object technology series)”. In: *J. Database Manag.* 10 (Jan. 1999).
- [CTV+19] Federico Ciccozzi, Matthias Tichy, Hans Vangheluwe, and Danny Weyns. “Blended modelling - what, why and how”. In: Sept. 2019, pp. 425–430. DOI: 10.1109/MODELS-C.2019.00068.
- [For22] Luca Forstner. *Integrating glsp based tooling into visual studio code*. Feb. 2022.
- [Fri21] Christoph Fricke. *Standalone web diagrams and lightweight plugins for web-ides such as visual studio code and theia*. Sept. 2021.
- [GB21] Philipp-Lorenz Glaser and Dominik Bork. “The bigger tool - hybrid textual and graphical modeling of entity relationships in vs code”. In: *2021 IEEE 25th International Enterprise Distributed Object Computing Workshop (EDOCW)*. 2021, pp. 337–340. DOI: 10.1109/EDOCW52865.2021.00066.
- [HDM+14] Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Loftus-Mercer, and Owen O’Brien. “Sccharts: sequentially constructive statecharts for safety-critical applications”. In: *ACM SIGPLAN Notices* 49 (June 2014), pp. 372–383. DOI: 10.1145/2666356.2594310.
- [HLF+22] Reinhard von Hanxleden, Edward A. Lee, Hauke Fuhrmann, Alexander Schulz-Rosengarten, Sören Domrös, Marten Lohstroh, Soroush Bateni, and Christian Menard. “Pragmatics twelve years later: a report on lingua franca”. In: *Leveraging Applications of Formal Methods, Verification and Validation. Software Engineering*. Ed. by Tiziana Margaria and Bernhard Steffen. Cham: Springer Nature Switzerland, 2022, pp. 60–89. ISBN: 978-3-031-19756-7.
- [HLL+18] Brian Hempel, Justin Lubin, Grace Lu, and Ravi Chugh. “Deuce: a lightweight user interface for structured editing”. In: *Proceedings of the 40th International Conference on Software Engineering. ICSE ’18*. Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 654–664. ISBN: 9781450356381. DOI: 10.1145/3180155.3180165.
- [KPE16] Sven Keidel, Wulf Pfeiffer, and Sebastian Erdweg. “The ide portability problem and its solution in monto”. In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering. SLE 2016*. Amsterdam, Netherlands: Association for Computing Machinery, 2016, pp. 152–162. ISBN: 9781450344470. DOI: 10.1145/2997364.2997368.
- [NK98] Stephen North and Eleftherios Koutsofios. “Applications of graph visualization”. In: (Mar. 1998).
- [Pan14] Arun K. Pande. “Dom manipulation”. In: *jQuery 2 Recipes: A Problem-Solution Approach*. Berkeley, CA: Apress, 2014, pp. 161–227. ISBN: 978-1-4302-6434-7. DOI: 10.1007/978-1-4302-6434-7\_6.

## Bibliography

- [Pet19] Jette Petzold. *Intentional layout in sprotty diagrams: defining user interaction*. Sept. 2019.
- [PH07] Steffen Prochnow and Reinhard von Hanxleden. "Statechart development beyond wysiwyg". In: Oct. 2007, pp. 635–649. ISBN: 978-3-540-75208-0. DOI: 10.1007/978-3-540-75209-7\_43.
- [Pro08] Steffen Prochnow. "Efficient development of complex statecharts". en. PhD thesis. 2008.
- [Ren18] Niklas Rentz. "Moving transient views from eclipse to web technologies". MA thesis. Universität zu Kiel, Nov. 2018.
- [RWC11] D. Rubel, J. Wren, and E. Clayberg. *The eclipse graphical editing framework (gef)*. Eclipse Series. Pearson Education, 2011. ISBN: 9780321718488.
- [Sch19] Connor Schönberner. *Intentional layout in sprotty diagrams: reevaluation gesetzter constraints*. Sept. 2019.
- [Sör18] Domrös Sören. "Moving model driven engineering from eclipse to web technologies". MA thesis. Universität zu Kiel, Nov. 2018.
- [Tsa18] Chun-Yen Tsai. "Improving students' understanding of basic programming concepts through visual programming language: the role of self-efficacy". In: *Computers in Human Behavior* 95 (Nov. 2018). DOI: 10.1016/j.chb.2018.11.038.