Institut für theoretische Informatik

Leibniz Universität Hannover

Bachelorarbeit

Tool for Generation of User-Defined Visualizations of Imperative Programs from Execution Traces

von Anton Lazarev

Matrikelnummer:
10014028

31. August 2020

Erstprüfer: Prof. Dr. Heribert Vollmer
Zweitprüfer: Dr. Maurice Chandoo

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst habe und keine anderen Hilfsmittel und Quellen als angegeben verwendet habe.

Hannover, den 31. August 2020

Anton Lazarev

Table of Contents

Preface **1**

1 Theoretical Background **2**
1.1 Systematic Approach to Programming 2
1.1.1 Traces 2
1.1.2 SPLib and SP Programs 6

2 Technical Background **9**
2.1 Haskell 9
2.2 HTML 10
2.3 CSS 11
2.4 JavaScript 11
2.4.1 jQuery 11
2.5 SVG 11
2.6 SP 11

3 Implementation **13**
3.1 Design 14
3.1.1 Control Bar 15
3.1.2 Control-Flow-Graph 15
3.1.3 Trace Table 17
3.1.4 Predicate-Tree 18
3.1.5 User-Defined Visualisation 19
3.2 Algorithms 21
3.3 Scope and Limitations 24

4 User Instructions **25**
4.1 Getting Started 25
4.1.1 Create Executable File 26
4.1.2 Create Traces from a SP File 26
4.2 First Visualisation 27
4.2.1 Create Visualisation without User-Defined Visualisations 27
4.3 Extended Visualisation 28
4.3.1 Create Visualisation with User-Defined Visualisations 28
4.3.2 Create User-Defined Visualisations 29

4.4 Errors 30
4.5 Using the Visualisation 31

5 Outlook **34**

List of Figures, Tables and Algorithms . . . . . **i**

Bibliography . . . . . **iv**

Preface

There is a lack of quick methods to justify the correctness of implementations of algorithms. This problem is further increased by the fact that most programmers do not write down their ideas but rather implement based on a mental representation. A mental representation allows executing the algorithm mentally for an input. Writing down any changes of variables and predicates for each execution step will result in a so-called execution trace or trace for short.
The systematic approach of generating traces as well from the mental representation as from the implementation for particular inputs allows finding deviations fast. Although proving the correctness still requires formal argumentation why the chosen input is sufficient. It is up to the user to find and explain the strategy for this. In addition to the approach a framework already exists simplifying this trace-programming approach. So far, the traces can only be represented in lists or spreadsheets which can be tedious to read sometimes.
The design and implementation of a tool that generates visualisation based on traces, source code and user-specific input is the core content of this thesis.
An appealing and comprehensible visualisation would further increase the usability of traces. For the visualisation of a trace the variables and predicates have to be put into context again. For this reason, the traces are visualised together with the programs control-flow and predicate-trees which are parsed from the source code of the program. A custom visualisation, which is designed for a specific program, would increase the quality of the visualisation by far. This last visualisation should be user-created and therefore allows unlimited possibilities for a visualisation of the same program. For example it could highlight visited nodes in a tree during the execution of a tree-search algorithm. Or the visualisation could remove edges from a graph step by step during an execution of an algorithm that looks for cycles in a graph.

1 Theoretical Background

1.1 Systematic Approach to Programming

The diversity of programming languages, programming approaches and programming styles makes it difficult to compare and examine them. Although most of them follow a similar approach. First, there is a mental representation of what the programmer wants to translate into a programming language and therefore to a computer program. This mental representation usually is an algorithm of some sort. There are ways to prove that an algorithm is correct. The formal verification uses mathematical techniques and reasoning to prove the consistency of the design of an algorithm with a precise definition of its functionality [Bje05]. This also means that the functionality has to be expressed mathematically or formal. Formalising a mental representation would mean writing any method, input, output and every object of that algorithm down, which contradicts a mental representation. Furthermore, there is only testing to check if the implementation from mental representation to program is correct. The systematic approach to programming offers a different solution here. Because there is already a mental representation of that algorithm, it is possible to execute it for a particular input. Testing a program focuses on only key aspects most of the time like input and output behaviour. This approach however focuses on a lot more. In addition to the input and output behaviour this approach takes the execution of an algorithm also into account. Observing changes within the execution for each execution step and writing down every change of a variable during an execution of that algorithm will result in a so-called execution trace or trace for short. It is quite easy to generate traces for different inputs using the mental model and then compare them to the traces of an implementation. Any deviation in the trace proves that the implementation differs from the mental model. [Cha18]

1.1.1 Traces

As mentioned before, traces or execution traces are a list of changes of variables during an execution of a program. This leaves the question on how to generate them. This will be demonstrated on a quite easy algorithm. The mentioned algorithm is designed to return the lowest index of an occurrence of a character in a string. If the string does not contain the character, the result is "-1". You can see the algorithm 1.1 written in pseudo code below using the following two helper functions. The first being $\#s$ which returns the length of the string s and the second being $s.charAtIndex(i)$ which will return the character of the string s at the index i. The codes's purpose is to make sure everyone

has the same mental representation of that algorithm. It is assumed that the algorithm is only available as a mental model.

Algorithm 1.1 Example algorithm: firstOccurrence

```
1: function FIRSTOCCURRENCE(s,c)
2:     i ← 0
3:     for i < #s do
4:         if s.charAtIndex(i) = c then return i
5:         i ← i + 1
6:     return -1
```

Table 1.1: Basic trace for Algorithm 1.1 with input: $s =$ "*aabbc*", $c =$'*b*'

line	s	c	i
1	"aabbc"	'b'	undefined
2	"aabbc"	'b'	0
3	"aabbc"	'b'	0
4	"aabbc"	'b'	0
5	"aabbc"	'b'	1
3	"aabbc"	'b'	1
4	"aabbc"	'b'	1
5	"aabbc"	'b'	2
3	"aabbc"	'b'	2
4	"aabbc"	'b'	2
HALT	"aabbc"	'b'	2

By going line by line through the code you can quite easily write down a basic trace as it was done in Table 1.1 for the input $s =$ "*aabbc*" and $c =$'*b*'. The first column describes the line of the algorithm in which the variables are changed while the following columns contain the respective values for the variables s, c and i. This trace creates a lot overhead and is therefore difficult to read by a human. The focus should lie only on changes of variables. By writing only these down, repetition, when writing down traces by hand will be reduced and the traces become also easier to read. In addition to that only the information which operation is active is required and not the exact line in the source code. Accordingly, consolidation of several lines also reduces the amount of space needed. There is information that is given in the code that is not reflected in the trace yet - the predicates. Keeping track of them further increases the information about the algorithm. Applying all these changes will generate the trace as seen in Table 1.2 for the same input.

Now, the first column represents the operation that is active. Operations correspond to one or more lines of code of the pseudo code in Algorithm 1.1. The additional last two columns show the values of the predicates at that operation. The algorithm starts with $INIT$ which represents the initialization and because s and c are never changed during

Table 1.2: Extended Trace for Algorithm 1.1 with input: $s =$ "$aabbc$", $c =$'b'

operation	arg1 (s)	arg2 (c)	i	$i < \#s$	$s.charAtIndex(i) = c$
$INIT$	"aabbc"	'b'	0	$True$	$False$
LOOP			1		
LOOP			2		$True$
SUCCESS					
HALT					

the execution, their values are only listed here. The initialization includes all code that is only visited once during the start of an execution. That is why it also includes the second line of the algorithm where the variable i is assigned the value 0. The second line describes the first iteration through the for-loop that only changes the value of i while the third line describes the second iteration. The first occurrence is found and the second predicates value changes to $True$. The success operation returns i but does not change any variable. $HALT$ is an artificial operation that represents the end of the program.
This new trace also offers additional information. Some operations can follow each other. This sequence is also called a control-flow which can be used to construct a control-flow-graph. A control-flow-graph is a directed graph which visually represents the sequence of operations that are taken within a program. In this example the sequence $INIT \rightarrow LOOP \rightarrow LOOP \rightarrow LOOP \rightarrow SUCCESS \rightarrow HALT$ can be constructed from the trace. The control-flow-graph in Figure 1.1 is generated by consolidating this sequence by the respective operations.

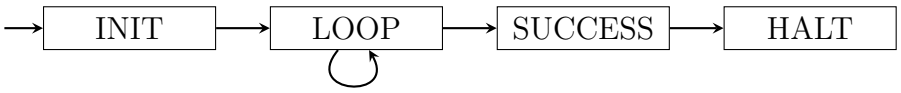

Figure 1.1: Control-flow-graph from the trace at Table 1.2

Assuming there are already several additional traces for that algorithm that also cover all the corner cases. A full control-flow-graph could be constructed from these. One information is still missing for a complete representation of an algorithm. At this point it is uncertain where to go if a control-flow branches out. In the algorithm this is described by the predicates. By creating binary decision trees for each operation in the control-flow-graph the entire algorithm flow would be represented by it as seen in Figure 1.2 and Figure 1.3. These binary decision trees are called predicate-trees and are trees with either a predicate or an operation as root. Operations do not have any children and therefore are also called leaves. Predicates always have two children and occur only in the inner nodes and the root. The predicates determine the path through a predicate-tree. If the predicate is $False$ the path follows the right child or it follows the left child when it is $True$. The path always must end in a leaf and hence an operation.
So, based only on a variety of traces it is possible to generate a complete control-

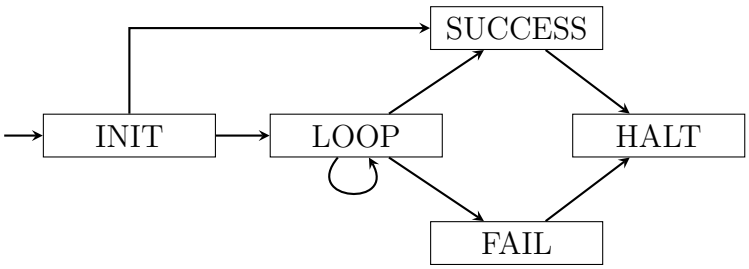

Figure 1.2: Full control flow graph for the Algorithm 1.1.

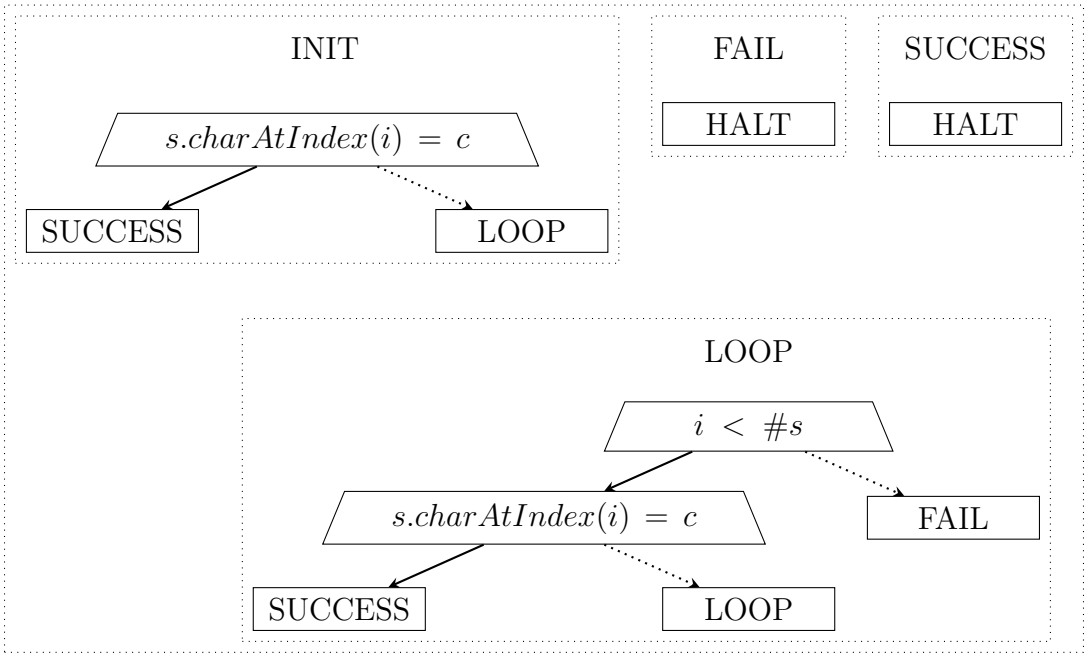

Figure 1.3: Predicate-trees for the Algorithm 1.1.

flow-graph as well a set of predicate-trees. The traces also offer information about the operations and what influence they have on variables and predicates. Therefore, traces allow a reverse-engineering of an algorithm even without knowing it beforehand. With the knowledge of the control-flow-graph (Figure 1.2) and the predicate-trees (Figure 1.3) and what happens to variables as well as predicates at each operation a programmer should have no problem to implement the algorithm in any imperative or functional programming language. However, this translation still might include mistakes and requires cognitive efforts. It takes even more effort with strictly functional programming languages which have limitations on side effects. Without side effects it is difficult to create traces which are obviously required to compare the implementation with the pre-existing traces. If the traces of the implementation coincide with the traces from the mental model for all inputs then the algorithm was implemented correctly. Testing for all inputs while the possibilities are infinite is of course impossible. As with other testing strategies the set of inputs has to be limited to cover all corner cases. This is

where having a control-flow-graph and all predicate-trees comes in handy. Several testing strategies can be applied here and depend on the complexity of the algorithm. With lower complexity not only visiting every operation but also covering all possible paths might be possible. With increasing complexity of the graph this approach becomes more and more impractical especially if loops exist. Complexity in a graph is defined by the number of nodes and the amount of edges. The more nodes and edges exist the more complex is a graph and also more paths from $INIT$ to $HALT$ exist. [Cha18]

1.1.2 SPLib and SP Programs

As mentioned before the translation from control-flow-graphs and predicate-trees to a programming language might lead to errors and increases cognitive load. This is where the SPLib framework comes in handy. The SPLib framework offers many features, one of them is the programming language SP. The SP language follows quite simple rules and allows to generate a program effortlessly from traces. The code of a SP program can be divided into several components that are themselves distinguishable from the source code by their headers marked with a # as seen in Algorithm 1.2.

SP code starts with $\#NAME$ that contains the name of the program. Followed by $\#FUNCS$ where all the different functions this program can offer are listed with function name, their input and respective output. In the example SP code, the function $fiOcc$ returns the index that is changed during the execution depending on the input. For demonstration purposes there is an additional function $fiOccStr$ that executes the same path as $fiOcc$ but returns s instead. Because s was never modified the input for s is returned. This unnecessary function only demonstrates that the same path taken through a program still might offer different functions when different variables are returned. A better example would be a program for division that returns an integer without rounding. The most obvious function returns an integer without rounding. An additional function could return the rest of that division.
The next components of the SP program are the $\#VARS$ which is basically a list of global defined variables that are used in the operations with their data type. The next component might seem a bit different from other programming languages. In $\#PREDS$ all the predicates used in the predicate-trees are defined. The next component are the operations in $\#OPS$. The operations are not only defined by name but also contain a Haskell like code that affects the variables. Last component is $\#FLOW$ which describes the control-flow within the program using the predicates and operations defined earlier. Assuming the algorithm contains an if-case with a logical condition. The if-case will not be written in $\#OPS$ at all. The logical condition will be defined in the $\#PREDS$. In the $\#FLOW$ section the predicate will be called and if it is true the first operation will be taken otherwise the second.
With a given set of traces, the corresponding control-flow-graph and set of predicate-trees a SP program can be transferred without much effort. The list of operations can be taken from the control-flow-graph while the flow can be directly transferred from all the predicate-trees. The variables and the functionality of an operation can be found in

Algorithm 1.2 Example SP code: firstOccurrence

```
#NAME
firstOccurrence

#FUNCS
fiOcc s c = i
fiOccStr s c = s

#VARS
s       :: String
c       :: Char
i       :: Int

#PREDS
p_length = ((length s) > (i))
p_charAt = ((s !! i) == c)

#OPS
o_init:
  i' = 0

o_for_loop:
  i' = (i+1)

o_success:

o_fail:
  i' = (-1)


#FLOW
o_init    = (p_charAt
                                  o_success
                                  o_for_loop)
o_for_loop = (p_length
                       (p_charAt
                                  o_success
                                  o_for_loop)
                       o_fail)
o_success    = HALT
o_fail       = HALT
```

the traces themselves.
The SP syntax for the operation resembles Haskell. The similarity is based on that the SPLib is programmed in Haskell. This fact allows a user to write helper-functions in Haskell and use them in the SP program. Thus, reducing the complexity of the SP program and this helps concentrating on the task at hand.
Compiled code from SP code is written in Haskell itself and can be run in a Haskell-Interpreter. A special feature of SP generated Haskell code is that it allows the output of traces.
Therefore, the framework allows creating code from traces with little cognitive effort and allows generation of traces from an algorithm.

2 Technical Background

The focus of this chapter will be on used technologies and programming languages. Only a few frameworks and ideas in the actual implementation are products of third parties and will be marked as such. The Figure 2.1 represents all the different inputs and programming languages that the visualisation tool needs to create an output.

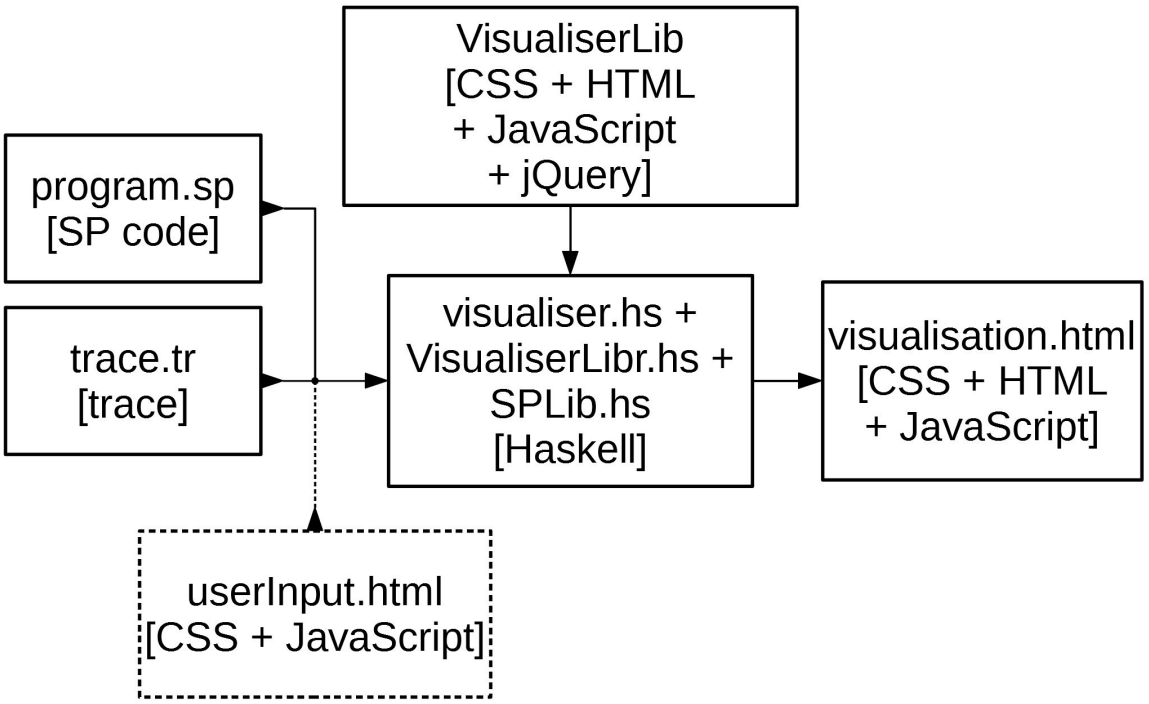

Figure 2.1: Used technologies and their connections.

2.1 Haskell

Haskell is a purely functional language. Many programming languages are imperative languages. This means they follow a sequence of commands that have to be executed in that exact order. In contrast to that a program in a functional language is not a sequence but a single expression that is evaluated on execution quite similar to a mathematical equation. This affects the stability of the language very positively and reduces

errors like side-effects. The stability of a language is a property that describes whether already implemented programs keep their functionality after updates or extensions to the language. A side-effect happens if a value of input or output is different in the same setup. Input from the environment like temperatures or time would be side-effects. In Haskell side-effects are limited to the use of monads only. During the implementation of the tool monads for IO had been used. Otherwise it would not have been possible to read in files or create any. Where many languages have while- or for-loops Haskell has to rely on recursive methods. Although its pattern matching and custom defined data structures make Haskell a very powerful and fast performing language.[Mar10]

2.2 HTML

HyperText Markup Language (HTML) is nowadays ubiquitous and plays a major role in the success of the *world wide webs*. Though on creation its purpose was not to design websites. HMTL was originally created to display text. The purpose shifted later. While books follow a linear layout, electronic text is not bound by this limitation. It can cross link, reference and be referenced. These features made HTML also very popular in the early websites. HTML alone can only change the layout of text but not the design. For this purpose, CSS was invented and together HTML and CSS are still very popular in website-design. Later versions of HTML supported images, videos and scripts. The current version is HTML5 which also was used.
HTML consists of elements that are also called tags. Creating nesting, ordering elements and using different types of elements helps in designing websites as seen in Figure 2.2. [BSS17]

```
<div class='content desktop-9-cols phone-9-cols'>
    <div class='flex-container2'>
        <div class='contents control_flow_graph' id='control_flow_graph'>
            <h3 class='headline_small'>
                Control-Flow-Graph (CFG) [draggable]
            </h3>
            <div id='container' class='container'>
            </div>
        </div>
        <div class='contents table_box' id='table'>
            <h3 class='headline_small'>
                Traces
            </h3>
        </div>
   ...
```

Figure 2.2: HTML-code example from *VisualiserLib/body.html*.

2.3 CSS

The *Cascading Style Sheets* (CSS) supports the HTML in many design aspects. With CSS it is possible to create styles for different objects, elements and classes. These styles can configure basic design features like the positioning and design of elements with padding, margins, font styles, colours and many more. More advanced features of CSS are the animations and the dynamic scaling of elements. The combination of basic and advanced features lets a website appear fluently and well designed. [BSS17]
The implementation of the visualisation tool relied on many CSS features even going so far that some visualisations were created purely by HTML and CSS alone. The predicate-tree visualisation was created by nesting of HTML and generating styles for the nested elements. This implementation was heavily based on a third-party contribution. [And17]

2.4 JavaScript

JavaScript was designed for supporting HTML and CSS in web design and it can be executed by many browsers. JavaScript is client based and does not require any internet connection. The purposes of JavaScript was to help out with forms, security and animation on the client side. Though the name might be miss-leading JavaScript is not object oriented or related to Java. As a script language it offers a lot of indulgence for mistakes.
JavaScript can be turned off in the browser which is a big disadvantage if a website relies on JavaScript. Turning off JavaScript is sometimes recommended due to malicious code on some websites. [BSS18]

2.4.1 jQuery

jQuery is a framework for JavaScript. As such it simplifies the use of some JavaScript features or offers new ones. The visualisation tool required the new coordinates of elements that were moved and jQuery offered a quite simple solution for that. [BSS18]

2.5 SVG

The *Scalable Vector Graphic* (SVG) is a graphic or picture that is dynamically created and can also be changed dynamically. It is described by code in an XML style. The advantages of SVG is that it describes a path rather than pixels. This way there is no quality loss when scaling the graphic. [FJJ00]

2.6 SP

The theoretical meaning of SP programs and the syntax were explained in Section 1.1.2 before. SP program are quite easy to grasp and also easy to create even without pro-

gramming skills. The only thing that a person needs is to create a control-flow-graph. Then they have to define what happens at each node and what determines the path if the control-flow has branches.
The framework then creates a working and executable program in Haskell. Other languages were supported before and might be supported in the future again.

3 Implementation

In the previous chapters of the visualisation tool the basic idea and the requirements had to be discussed. After that, the major requirements were prioritized based on importance for the tool and how crucial they were for further progression. Some requirements specified the product of the tool, which means the entire visualisation for one specific SP program and trace, while other requirements specified the tool itself.

- [**R0**] The visualisation tool should be able to construct a visualisation based on a SP program and an execution trace.
- [**R1**] The product of the visualisation tool has to show the control-flow-graph generated from traces.
- [**R2**] The product of the visualisation tool has to show the corresponding predicate-tree to the current operation.
- [**R3**] The product of the visualisation tool would have to show the if variables had changed in the current operation compared to the variables of the last operation.
- [**R4**] The product of the visualisation tool should visualise the SP program by visualising all the traces step by step.
- [**R5**] The product of the visualisation tool should also handle a trace file with more than one function call.
- [**R6**] The visualisation should offer intuitively information rather than overwhelm the user.
- [**R7**] The product of the visualisation tool should be contained in one file only and has to work standalone, therefore it is advisable to use HTML, CSS and JavaScript.
- [**R8**] The visualisation tool should use the framework around SPLib, hence it is advisable to be written in Haskell.
- [**R9**] The visualisation tool should be able to construct a user-defined visualisation based on a SP program, an execution trace and a user constructed template.

In summary the final product should be a tool written in Haskell that can create a HTML containing CSS and JavaScript to visualise a SP program and a corresponding trace. In addition to that a user should be able to program a user-defined visualisation that the tool should also take as an optional third input. This special feature would not limit the visualisation tool only to sorting-problems or tree-searches but it would allow to visualise every program if provided with the proper user-defined program.

3.1 Design

The implementation and the design process were done in several iterations. These iterations ended when some milestones were met. Then the iteration was presented and the feedback was implemented in the next iteration in addition to a new milestone. In Figure 3.1 you can see one of the earlier iterations for the design. It shows the layout of key components for the visualisation. To left there is the control bar, to the bottom the footer and in the centre the area was divided in four sectors. These sectors contain the control-flow-graph, the trace table, the predicate-tree and the user-defined visualisation.

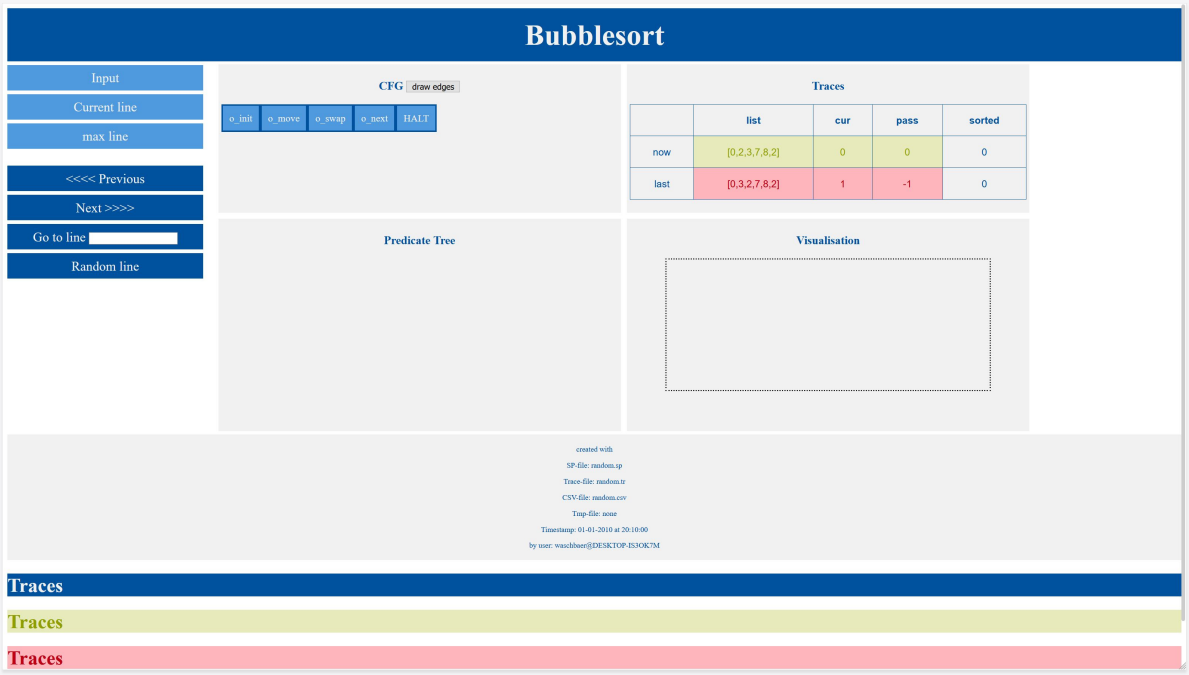

Figure 3.1: Early design of the layout.

In the beginning the designing process was solely made in HTML and CSS rather than sketches or other designing software. This was a deliberate decision and offered many advantages. The design could be tested on different displays and browsers similar to the final product. Also, the produced code could be reused in the later iterations and the final product. The focus of the early design iterations was on the layout of the four different visualisations and a matching yet pleasing colour theme.

3.1.1 Control Bar

Because the visualisation of the traces should be dynamic, controls for navigation were needed. If the navigation controls are not planned properly it can become very confusing for a user while navigating. That is why already in the early design iterations the control bar contained information about the *current line* and *max line* of the trace. This way the sensation of feeling lost should be prevented. In addition to orientation the design of the control bar offered an intuitive distinction between information elements and push-able buttons by grouping them together and colouring them differently. By using the mouse-over feature of CSS direct feedback could be implemented and increase the usability of the interface. The final design of the control bar still featured a lot of the early designs but grew on them.
In the final design a help section was implemented which holds information about different aspects of the visualisation. The information is displayed as a tool-tip on mouse over hence reducing the need of another layout design.

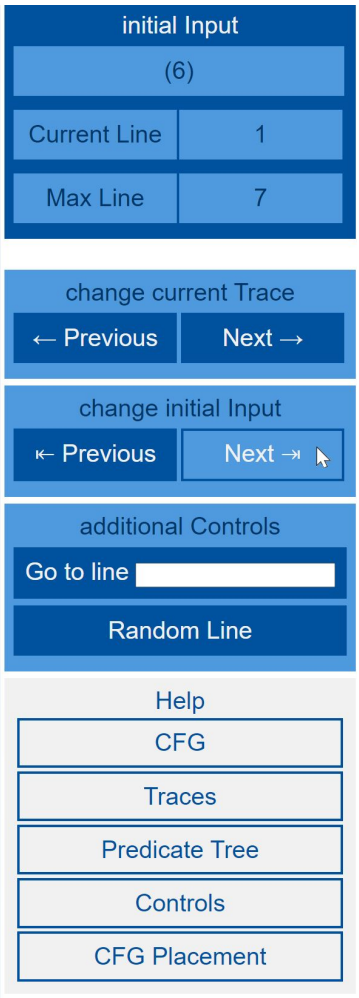

Figure 3.2: Final design of the control bar.

3.1.2 Control-Flow-Graph

The visualisation of the control-flow-graph is not only very important but also very tricky. The visualisation tool should work on every SP program and therefore there is no limit to the amount of operations defined in the SP code. Thus, making the calculation for placing the elements that represent operations in the control-flow-graph rather difficult. But the control-flow-graph should not only visualise the operation but also the flow accordingly the edges between elements. Imagine a rather basic placing algorithm for elements not taking the edges in consideration. As a result of this the visualisation has a lot overlap and for that reason a very bad visualisation effect. A proper algo-

rithm was needed to place elements cleaner and with less overlap regarding the edges. The idea was to place the largest clique of the graph first, then arrange all other nodes in descending order of their edges. A clique in a graph is a subset of nodes that are all connected to each other. But finding a clique with at least n elements is a known NP-complete problem and therefore its calculation time is increased severely for every element in the graph [MV15]. Keeping this in mind the design of the control-flow-graph

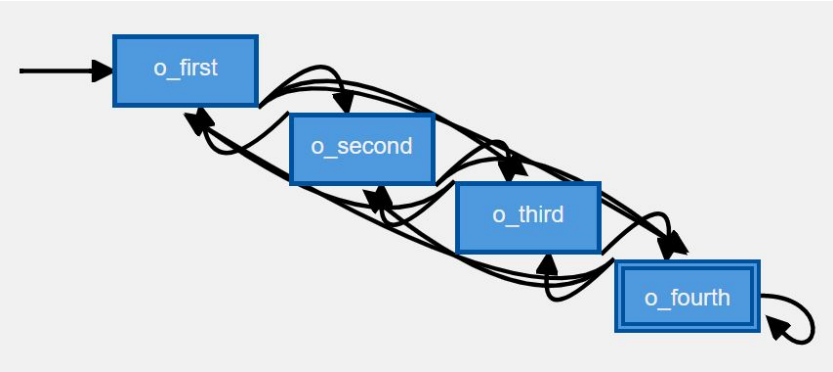

Figure 3.3: Basic placement of elements with maximum edges.

was steered early to user-defined placement. This feature would allow a user to place the elements in the optimal way. For dynamically placing elements the drag and drop feature of JavaScript for HTML-elements was used. Though moving elements by drag and drop is rather a standard feature of JavaScript, but getting the coordinates of that moved element is not. For this functionality jQuery a JavaScript framework had to be imported. The available coordinates were used to generate the edges. Rather relying on any library and for this reason increasing dependencies of the tool the algorithms for drawing and redrawing of the edges were implemented custom-made, see Figure 3.4.
The ability to save and share the placement of the control-flow-graph has been implemented to further increase the usability. The coordinates were stored in strings which were stored in the local storage of the browser. This way the placement is saved even across sessions.

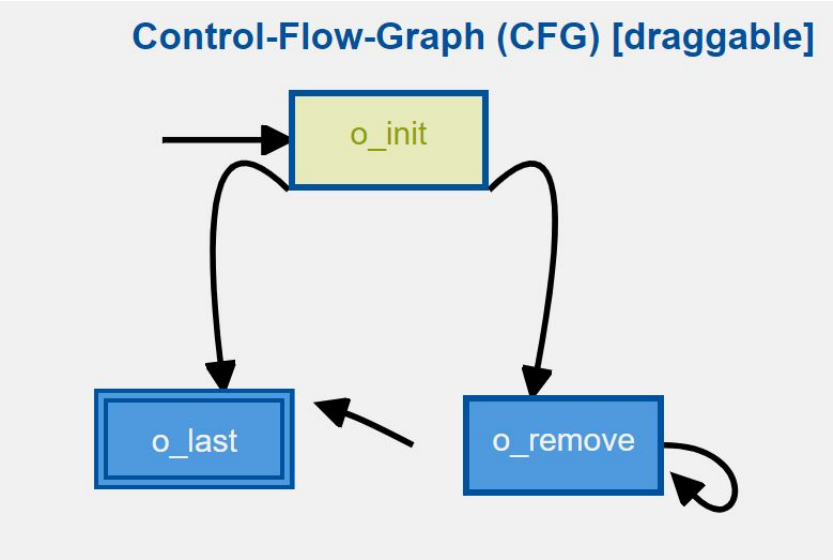

Figure 3.4: Final design of the control-flow-graph visualisation.

3.1.3 Trace Table

The design of the trace table was rather straight forward. Each column represents a variable and the first row represents the value of the corresponding variable. The second row shows the value of the last operation in the trace. If the value has changed it was highlighted by colour. The highlighting by colour was discarded during one of the design iterations due to the numerous uses of colours in the other visualisations. Instead of colouring the cells now merging them together was an indicator for unchanged values, see Figure 3.5.

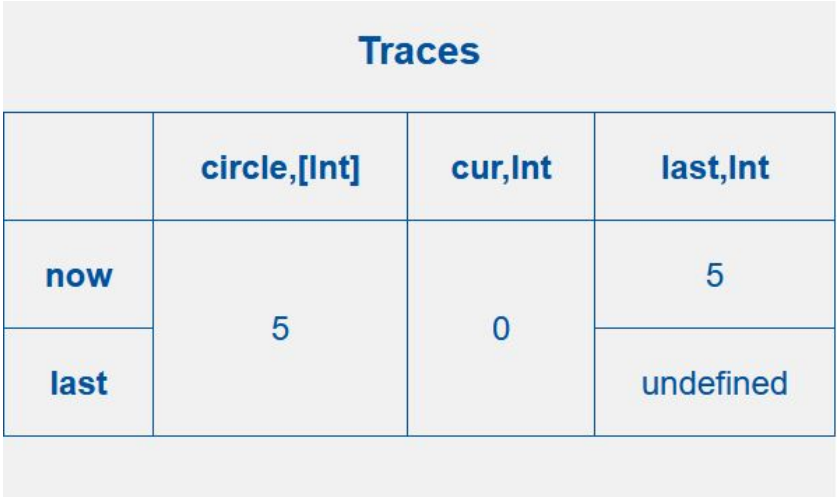

Figure 3.5: Final design of the trace table.

3.1.4 Predicate-Tree

During the implementation of the visualisation tool the control-flow-graph also offered coloured arrows to indicate what the previous operation was and what the future operation will be. After implementation and testing this feature it was discarded for the same reasons as the colouring in the trace table. As a result the visualisation of the predicate-tree is the only source of information about the future operation. A neat and appealing visualisation of binary trees is quite challenging. Furthermore, sorting the children of an unordered tree to minimize the width is a NP-complete problem [MS04]. Research has found a great solution for binary trees. While the visualisation is neatly and intuitive the required code is perfect for the indented use. The design of the predicate-trees was inspired a lot by the work of Andrean [And17]. The simplicity only using HTML nesting and CSS to create a binary tree made it much easier to generate predicate-trees dynamically. Styles and positions are taken care of only by creating the correct nesting. In addition to the layout of the binary tree the predicate-trees should also offer information about the values of the predicates and the future operation. As a reminder operations are all leaves in the tree. The next operation is determined by a path from the root to any leaf. Visited predicates are either $True$ or $False$ and by colouring them green or red an intuitive visualisation is achieved. Not visited nodes are not recoloured. The next operation is highlighted in a distinct colour.

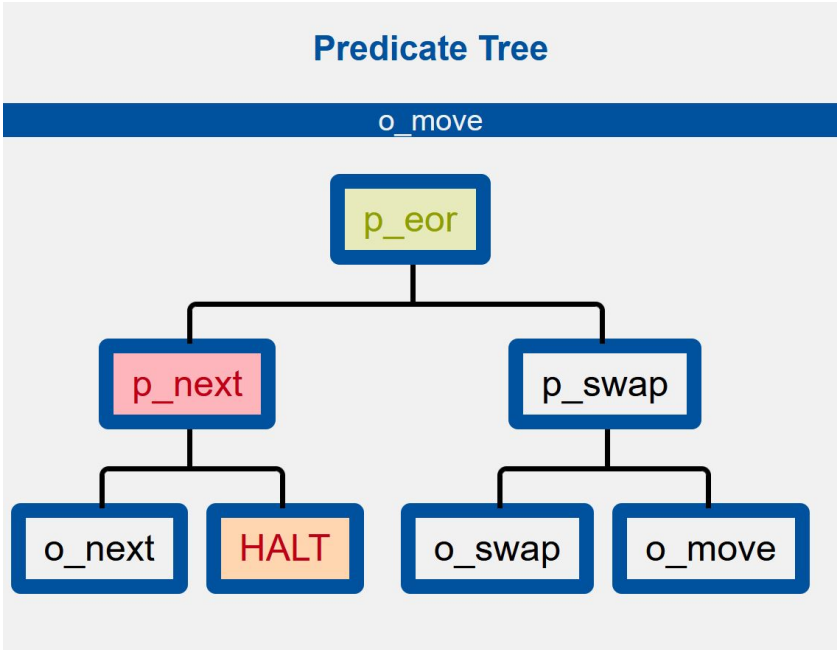

Figure 3.6: Final design of the predicate-tree.

3.1.5 User-Defined Visualisation

The user-defined visualisation is in a distant way a kind of interface which offers full customisation. This means if any users want a specific visualisation for a SP program, they can implement one by themselves. Therefore, this feature offers up to unlimited visualisations and visualisation styles. Creating something completely new is difficult. Accordingly, two example user-defined visualisation are already implemented and the interface is intuitive and well documented.

Bubblesort

Bubblesort is one of the many simple sorting algorithms. The most common visualisation for a sorting algorithm is a bar diagram. The positioning along the x-axis represents the position in the list or data-structure and the height of the bar, or y-axis value, represents the value of the list element. During the execution of a sorting algorithm the bars will be sorted either ascending or descending based on the way the algorithm was implemented.

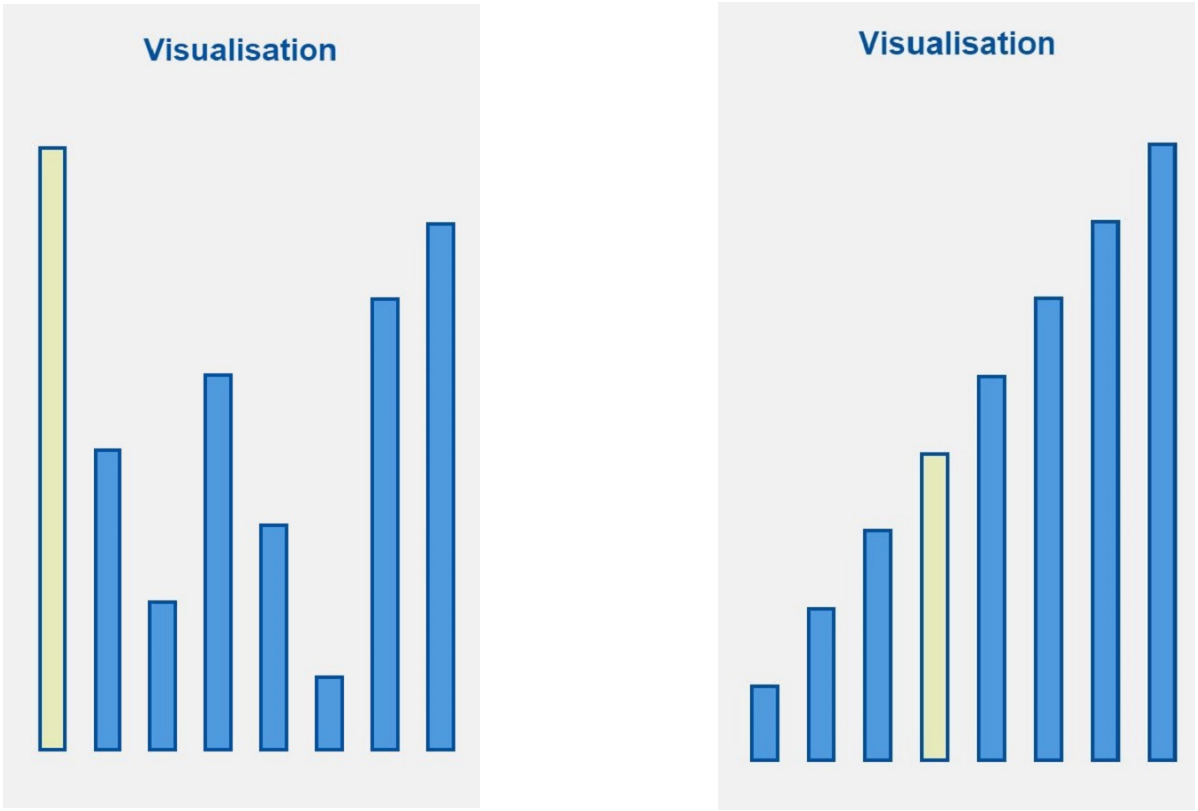

Figure 3.7: Final design of the user-defined visualisation for bubblesort for an unsorted list (left) and a sorted list (right).

Josephus Problem

The Josephus problem is a counting-out problem with a story behind it. 40 Jewish soldiers were about to be captured by romans soldiers and they chose death over capture. Josephus was one of them but did not want to die. They arranged themselves in a circle. From a starting point or rather the starting soldier an action sequence was executed. The first person kills his left neighbour. The next living person in line kills his left neighbour. This sequence is continued until only one person is left living in the circle. Josephus wanted to be that man. The question was where he had to stand to survive. The idea for the visualisation of the Jospehus problem is given by the story. A number of nodes are arranged in a circle and are enumerated. The active person is highlighted while the dead are marked as such.

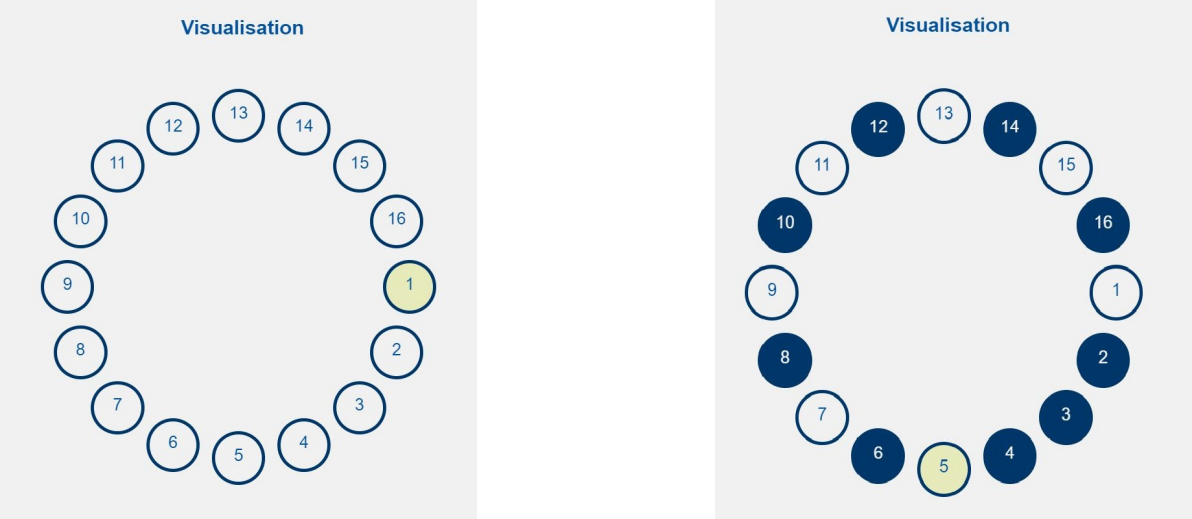

Figure 3.8: Final design of the user-defined visualisation for the josephus problem at the beginning (left) and during the execution of the algorithm (right).

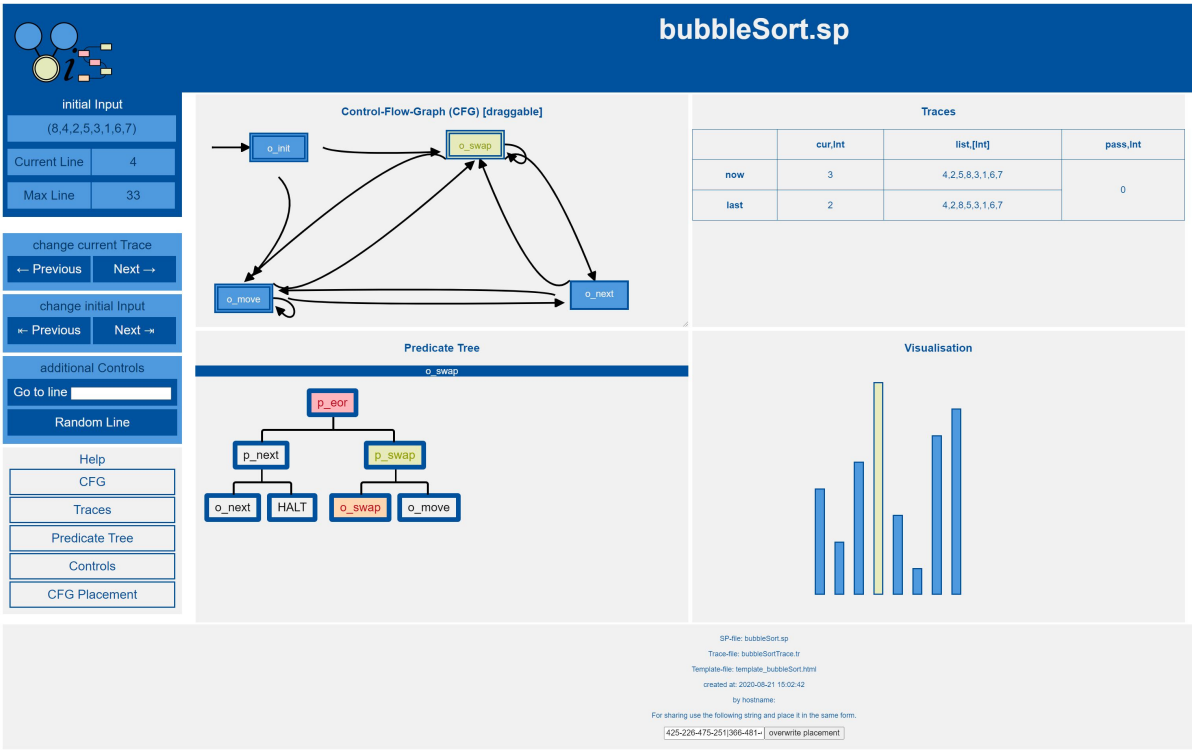

Figure 3.9: Final design of the layout.

3.2 Algorithms

The earlier section described the design and functionality of the front-end. The focus of this section is on the back-end and some of its major algorithms. Before that the creation of the HTML file is explained first. In the early designing iterations, a state was reached where the information of the traces was needed to progress further. This was the turning point where most of the work was done in the back-end. The resulting HTML file consists of several sections which are sometimes also segmented themselves. The head section contains JavaScript, the next section styles contains CSS. The body section follows, containing HTML and JavaScript. Many lines of code could be hard-coded in Haskell. For maintainability reasons they were outsourced to other files in the $VisualiserLib$ directory. The body section of the HTML code is a copy from the *body.html* file. The script section has several areas where the code is either statically copied from the *scripts.html* or dynamically generated. The CSS section is static but can be a combination from $visualiser - CSS.css$ and a *template.html*. All sections then are weaved into one final HTML file containing HTML-code, CSS-code, copied JavaScript-code and dynamically generated JavaScript-code as can be seen in the Figure 3.10 and Figure 3.11.

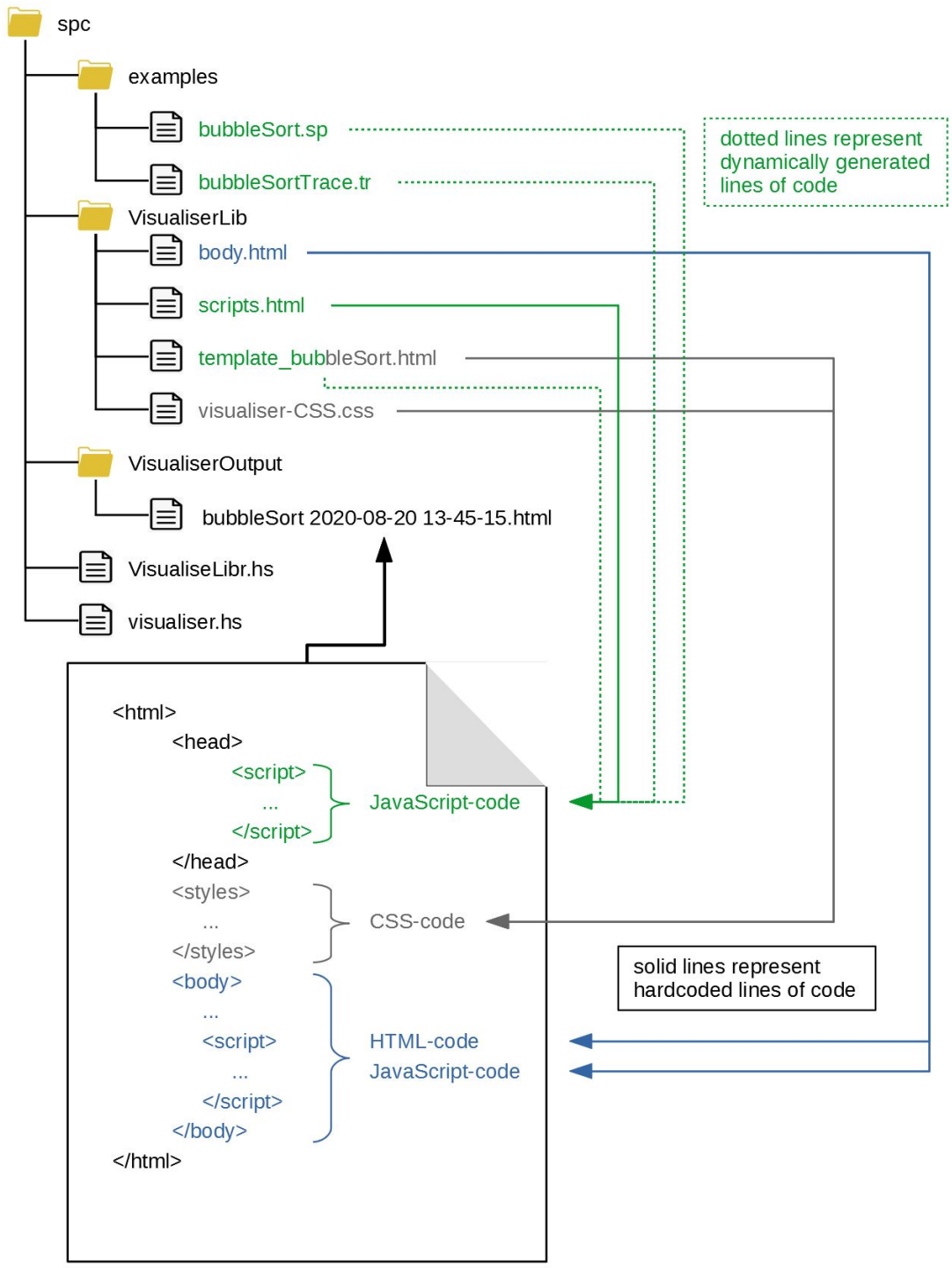

Figure 3.10: Representation of the influences of input files in the final product when generating a visualisation.

```
VARS.push(["cur","Int"]);
VARS.push(["list","[Int]"]);
VARS.push(["pass","Int"]);
VARS.push(["sorted","Bool"]);
PREDS.push(["p_eor","((cur + pass + 1) == (length list))"]);
PREDS.push(["p_ls2","(largerThan (length list) 2)"]);
PREDS.push(["p_next","((not sorted) && ((pass + 2) < (length
↪  list)))"]);
PREDS.push(["p_swap","((list!!cur) > (list!!(cur+1)))"]);
FLOW.push(["o_init"]);
FLOW.push(["o_move"]);
FLOW.push(["o_next"]);
FLOW.push(["o_swap"]);
OPS.push(["o_init",[("cur'","0"),("pass'","0"),("sorted'","True")]]);
```

Figure 3.11: Example dynamically generated JavaScript code.

SP Code and Traces

The SPLib framework offered methods to parse SP code and traces. The SPLib framework has defined several data structures in Haskell to simplify working with SP code or traces. Some methods were needed to extract data from these data structures. While Haskell can handle custom data structures JavaScript cannot. The solution was to use arrays instead. Arrays in JavaScript can be changed in their size dynamically. Several methods were implemented in Haskell to loop over SP components and simultaneously create strings. These strings are a result of concatenating hard-coded strings with information from the components. They are JavaScript commands to fill the corresponding array with the information from the trace or from the SP components. These commands are all called in the *start*() function, which is executed at the start, in the resulting HTML.

Dynamically created Functions

For some mechanics weaving lines of code within a function was not enough. Entire functions had to be generated dynamically. Algorithm 3.3 is an example for this. For each element of the control-flow-graph a similar function has to be defined, otherwise the coordinates of a dragged element cannot be assigned correctly. Another example are the functions that create the predicate-tree.

Control-Flow-Graph Edges

The single most time-consuming method to implement was the method to generate the edges in the control-flow-graph. While there are libraries that offer similar functions,

Algorithm 3.3 Example algorithm for one of the contron-flow-graph elements that are draggable.

```
$( function() {
        var element = $( '[id=op3]');
        $( '[id=op3]' ).draggable({
                start: function() {
                dragged(element);
                },
                stop: function() {
                dragged( element );
                drawEdges();
                }
        });
 } );
```

none were exactly what was needed. Therefore, the entire visualisation was implemented independently. The idea was to calculate the starting points and entry points for the arrows. In addition to that Bezier curves, which are a slightly curved connection between two points, were used to distinguish the direction more and reduce overlap. The arrows themselves are SVG elements that follow a specific path. The starting and finishing points were calculated on the angle and placements of the elements to each other.

3.3 Scope and Limitations

Basically, for all SP program their trace can be visualised. The visualisation tool can handle traces with many different inputs and can even generate visualisation for SP programs with a lot of operations. The limitations of the tool are not in generating code but rather the visualisation itself is limited.
The trace table can handle a lot of different variables due to the implementation of a scroll bar. But it is not a very convenient visualisation if you must scroll back and forth to see the information you need. In addition, the display of the control-flow-graph has its limitation when it comes to the amount of nodes and edges. The drawing space could be increased but that would not benefit any visualisation.
The program and function calls derived from the trace and the SP file but the subprogram calls are not. Accordingly, a visualisation of subprogram calls would require further input.

4 User Instructions

The visualisation tool is written in Haskell and requires the Glasgow Haskell Compiler (GHC) to be compiled. The current Version 8.10 (released on 8-8-2020) is supported by the most common operating systems. The latest version is not required but earlier versions might have issues with compiling due to hidden dependencies mostly concerning the *Data.Map* dependency. If such error occurs updating to the latest GHC version or using a different way of installation (cabal, chocolate, stack) might help.

4.1 Getting Started

Before any visualisation can be created the executable file has to be compiled. First, it is important to make sure that the visualisation tool is unpacked in the same directory as *SPLib.hs*. The visualisation framework contains the following files:

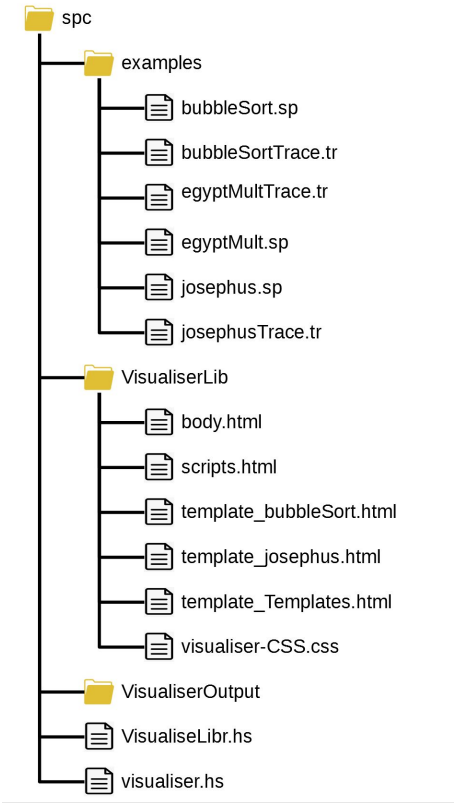

Figure 4.1: File tree representing all the files related to the visualisation tool

4.1.1 Create Executable File

The creation of an executable file is simple and only consists of two steps.

1. Open a terminal in the path *../spc/*.
2. Compile the *visualiser.hs* file with the Glasgow Haskell Compiler.

```
$ ghc ./visualiser.hs
```

Compiling might take some time and depending on your operating system you should get a *visualiser* or *visualiser.exe* as a result.

4.1.2 Create Traces from a SP File

The creation of traces from a SP file consists of a few steps and does require not only the *SPLib.hs* library but also the *SPLib* directory. The files contained in this directory make it easier to generate traces. The step by step guide explains how traces are created for the example algorithm *firstOccurrence*.

1. Place the SP file next to the SPLib directory.
2. Open a terminal of your choice and compile the SP file.

```
$ spc .\firstOccurence.sp
```

3. Open the compiled Haskell file with a text editor of your choice and add two lines at the very top. It should look like this:

```
import SPLib.Basic
import SPLib.List

--START OF PROGRAM firstOccurrence
...
```

   Without these imports the trace functions that are weaved in during the compilation of the SP file will not work.

4. Open the file *SPLib/Basic.hs* with a text editor of your choice and add your program name to the whitelist at the very bottom of that file.

```
...

shouldProgramBeTraced pName = elem pName whiteList
  where
    whiteList = ["firstOccurrence",...]
```

   The generation of the trace is now whitelisted and ready to be created.

5. Open the compiled Haskell file with the Glasgow Haskell Compile Interpreter (GHCI) while piping the error-output to a specified file, that serves as an output for the traces.

```
$ ghci .\firstOccurence.hs 2>firstOccurenceTrace.tr
```

6. While using the interpreter you can call your methods and functions defined in the SP file and while they are executed the trace will be piped to the output. It is possible to make several function calls within one trace. After completion exit the interpreter with the command ":q" .

```
GHCi, version 8.10.1: https://www.haskell.org/ghc/  :? for help
[1 of 4] Compiling SPLib.PartialFunction ( SPLib\PartialFunction.hs,
↪  interpreted )
[2 of 4] Compiling SPLib.Basic      ( SPLib\Basic.hs, interpreted )
[3 of 4] Compiling SPLib.List       ( SPLib\List.hs, interpreted )
[4 of 4] Compiling Main             ( firstOccurence.hs, interpreted )
Ok, four modules loaded.
*Main> f0 "aab" 'b'
2
*Main> f= "aab" 'a'
0
*Main>:q
Leaving GHCi.
```

7. The file *firstOccurenceTrace.tr* contains now the traces for the inputs $s =$ "*aab*", $c =$'*b*' and $s =$ "*aab*", $c =$'*a*'.

For your own SP program some steps have to be modified especially the functions that are executed in the interpreter.

4.2 First Visualisation

With a working SP program and a created trace, it is now possible to generate the first visualisation.

4.2.1 Create Visualisation without User-Defined Visualisations

The creation of the visualisations is rather simple and only takes two steps.

1. Open a terminal of your choice and execute the following command.
   - On Linux:

```
$ .\visualiser <Path to the SP file> <Path to the trace>
```

- On Windows:

```
$ .\visualiser.exe <Path to the SP file> <Path to the trace>
```

- Example on Windows:

```
$ .\visualiser.exe .\firstOccurence.sp .\firstOccurenceTrace.tr
```

2. Navigate to the directory *VisualiserOutput* and look for the file that contains the SP program name with the newest timestamp and open it in any browser.

4.3 Extended Visualisation

This section is about creating your very own visualisations or creating an extended visualisation with the provided two examples.

4.3.1 Create Visualisation with User-Defined Visualisations

The creation of a visualisation with user-defined visualisations is quite similar to the creation without the user-defined visualisation. The only difference is the amount of arguments that the executable file gets on execution.

1. Open a terminal of your choice and execute the following command.
   - On Linux:

```
$ .\visualiser <Path to the SP file> <Path to the trace> <Path to the
↪  template.html file>
```

   - On Windows:

```
$ .\visualiser.exe <Path to the SP file> <Path to the trace>  <Path to
↪  the template.html file>
```

   - Example on Windows:

```
$ .\visualiser.exe .\examples\bubbleSort.sp
↪  .\examples\bubbleSortTrace.tr
↪  .\VisualiserLib\template_bubbleSort.html
```

2. Navigate to the directory *VisualiserOutput* and look for the file that contains the SP program name with the newest timestamp and open it in any browser.

4.3.2 Create User-Defined Visualisations

The creation of user-defined visualisations requires advanced knowledge with HTML, CSS and JavaScript. The file *template_Templates.html* serves as a template for user-defined visualisations. It consists of two areas divided by an @-symbol. The upper area is what the visualisation tool will weave in the CSS section of the resulting HTML file. There are also some references for the used colours in the general visualisation found in the comments.
The second area is what the visualisation tool will weave in the script section of the resulting HMTL file. The function *drawTemplate* will be called on every redraw of the visualisation if a user-defined template was provided. This function has to be expanded for user-defined visualisations. Creating HTML elements, styling them with CSS and appending them to the visualisation canvas is a quick and easy way to generate visualisations. Because the visualisation is depending on the variables in the traces it is recommended to create a visualisation first without user-defined templates. Inspect the HTML source code. Not indented lines in the script section of that HTML are the interesting dynamically created lines of code. The variables can be taken from a multi-dimensional array called *TraceList* as shown in Figure 4.3.
The first dimension of this array distinguishes several function calls with different inputs in the same trace. For a trace only having one call this dimension would always be 0 for the first input. The second dimension is similar to the first one but distinguishes subprogram calls. The visualisation tool cannot visualise subprogram calls yet but the traces are parsed in a way that it could become possible one day. The third dimension represents the trace line number within the program number and subprogram number. The fourth dimension now contains the trace itself. It is highly recommended to look at the source code of the HTML because the trace is an array itself and contains program names, arrays for the variables and arrays for the predicates hence the position of a particular variable is not intuitive.

```
1 TraceList[0][0].push(['o_for_loop',['c',3,"aabc"],[['p_length',True],
  ↪  ['p_charAt',True]]]);
```

Figure 4.2: Example line in source code containing TraceList.

Assuming the user-defined visualisation needs the string variable based on the example source code in figure 4.2 the solution would look like this. This should help finding the

```
1     var string_variable = TraceList[0][0][lineNumber][1][2];
```

Figure 4.3: Example reference to a variable in the trace.

needed information from the traces. Pretty much any visualisation can be created and weaved in the final HTML by the visualisation tool.

4.4 Errors

Some error handling has already been implemented. The amount of arguments and the file extensions are checked. Error messages for too many or too few arguments and for wrong file extensions can be displayed.

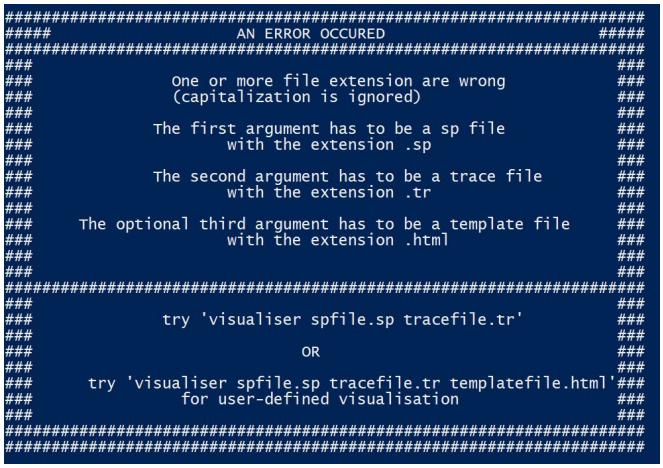

Figure 4.4: Error message for wrong file extensions.

- **There are errors about variables being not in scope when compiling the visualiser execution file.**
  Solution: The GHC version is either outdated or uses hidden dependencies. Try updating GHC or switch to another installation of GHC (cabal, stack, chocolate).
- **There are a lot of errors when compiling the visualiser execution file.**
  Solution: Did you compile the *visualiser.hs* file or maybe by mistake another file like *visualiser.exe*.
- **The generated trace is empty.**
  Solution: The SP program was not whitelisted in the file *SPLib/Basic.hs* or no function was called in GHCI.
- **The trace contains additional error output and which makes it unusable.**
  Solution: This might happen if you use the GHCI in the PowerShell on Windows, try using GHCI in the *cmd* instead.
- **The trace file is read-only.**
  Solution: You did not exit the GHCI and are still piping to the trace file. Use : q to exit.
- **The visualisation tool says that a file does not exist.**
  Solution: Check if the file exists and the path is correct.
- **On creation of a visualisation there is a CallStack error**
  Solution: Check if the files indeed are a SP file and and a trace file.
- **The visualisation has no input or next traces.**
  Solution: The trace file is either empty or maybe got mixed up by a trace belonging to another SP program.

4.5 Using the Visualisation

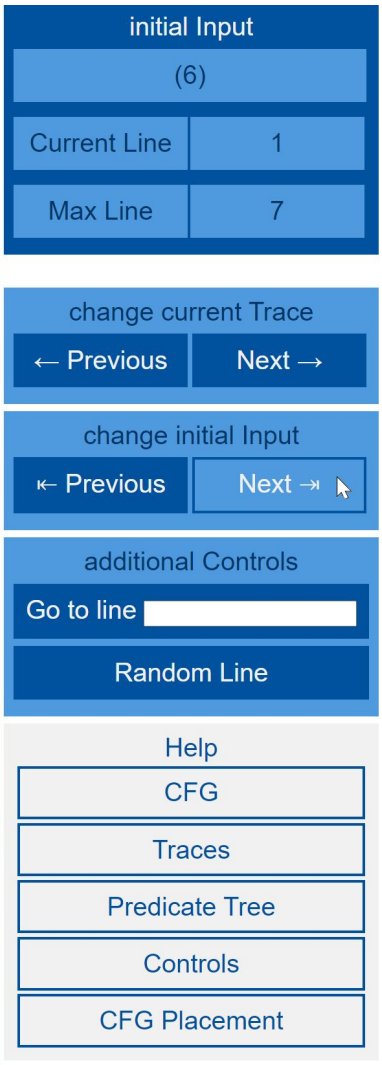

Figure 4.5: Control bar.

The control bar has three sections. The top section offers information about the initial input and the current position of the visualisation.
The middle section offers buttons to either skip to the next trace line or even skip to the next initial input. The button *Go to line* allows you to skip to a specific line. The button *RandomLine* makes you jump randomly.
The bottom section offers help by tool-tips.

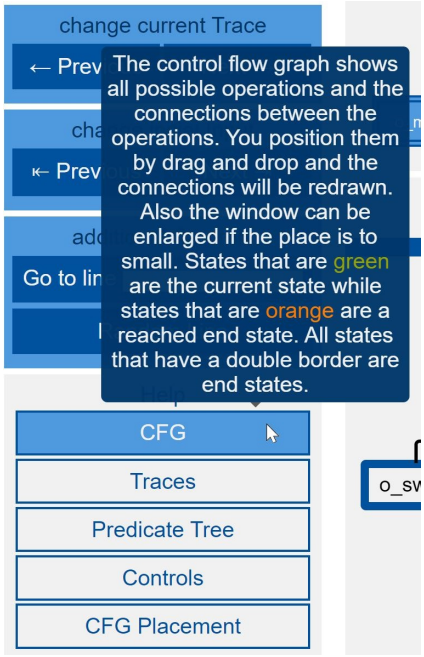

Figure 4.6: Example of a tooltip.

The footer contains several information about the visualisation. The names of the used files and the creation date are listed here. Also, the input form for sharing the placement of the elements in the control-flow-graph is here. For sharing just copy the entire string and share it. The receiver of that string only has to paste it in the same form and confirm by clicking the button *overwrite placement* (see figure 4.7.

Figure 4.7: Footer with information.

The elements in the control-flow-graph can be dragged and the arrows will be redrawn. The green highlighting of the nodes represents the currently active operation while an orange highlighted node represents a final operation. The starting operation is indicated by the single arrow to the left while a control-flow to the same operation is indicated by a curved arrow to the same element on the right. Nodes with double borders mark operations that can be final. The canvas can be resized if the amount of space is too small for arranging the elements. To do so drag the bottom right corner of the control-flow-graph.

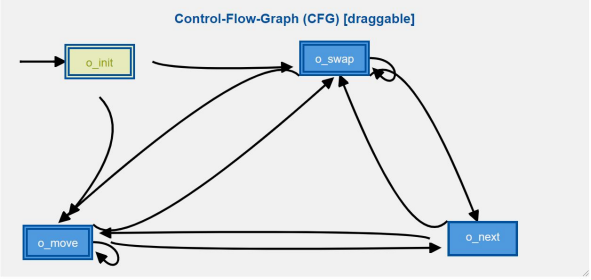

Figure 4.8: Control-flow-graph with dragged elements.

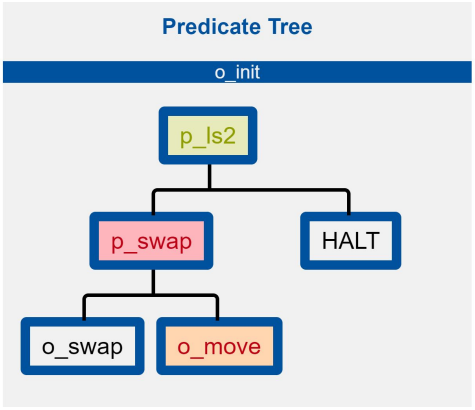

Figure 4.9: Predicate-Tree

The predicate-trees shows the binary decision tree for the currently active operation. Nodes with green highlight are predicates that were *True* and therefore the left path was taken. With red highlighting the predicate was *False* and the right path was taken. Orange highlighting indicates the next operation that will be taken.

The trace table shows the values of the variables for the currently active operation and the operation before that. Merged cells indicate that the values did not change.

Traces

	cur,Int	list,[Int]	pass,Int
now	1	1,2,3,4	0
last	0		

Figure 4.10: Trace table.

5 Outlook

The visualisation tool does support the use of SP code and traces. With the step by step visualisation that shows what happens during an execution of the program, it is easy to see if a step differs from the mental representation. Therefore, it assists in debugging SP programs. It proved its usefulness while debugging the $firstOccurence.sp$.

Subprogram Visualisation

The current version of the visualisation tool does not allow visualisations of subprogram-calls yet. The solution approach of parsing the traces to JavaScript already included this possibility. Only a few changes in the *visualiser.hs* have to be made to allow additional SP files as additional arguments. Parsing the information contained in them to similar arrays as the main SP file, f.e. $sub0 - VARS$, $sub0 - PREDS, ...$ would allow handling of multiple SP files in JavaScript. For usability reasons the controls have to be adjusted to allow to either skip a subprogram call or to show it.

Additional Templates

A more abundant collections of templates would not only increase the effect of visualisations but also simplify the creation of additional templates. A further goal would be a generalisation of similar visualisations. This would require some specifications of how a SP program has to look and work like so that the visualisation can be applied. This goal could be achieved by creating some dummy predicates and variables and specifying what variables are important for that user-defined visualisation.
A general visualisation for sorting algorithms could be achieved this way. Sorting algorithms like *quickSort*, *mergeSort*, *bubbleSort* or even *randomSort/bogoSort* do not share the same number of variables or predicates but they share the same information needed for a sorting algorithm visualisation. Implementing SP programs for the five mentioned sorting algorithms using ten dummy variables and predicates each and saving the current list to the tenth variable, could offer a sort of interface for a more generic visualisation. Like mentioned earlier, the most common visualisation for a sorting algorithm is a bar diagram like in Figure 3.7. Accordingly, a bar diagram with additional sorting algorithm specific information would be a good visualisation. For example, the specific information would visualise the number of lists in *mergeSort* or the pivoting index in *quickSort*.

List of Figures

1.1 Control-flow-graph from the trace at Table 1.2 4
1.2 Full control flow graph for the Algorithm 1.1. 5
1.3 Predicate-trees for the Algorithm 1.1. 5

2.1 Used technologies and their connections. 9
2.2 HTML-code example from *VisualiserLib/body.html*. 10

3.1 Early design of the layout. 14
3.2 Final design of the control bar. 15
3.3 Basic placement of elements with maximum edges. 16
3.4 Final design of the control-flow-graph visualisation. 17
3.5 Final design of the trace table. 17
3.6 Final design of the predicate-tree. 18
3.7 Final design of the user-defined visualisation for bubblesort for an unsorted list (left) and a sorted list (right). 19
3.8 Final design of the user-defined visualisation for the josephus problem at the beginning (left) and during the execution of the algorithm (right). . 20
3.9 Final design of the layout. 21
3.10 Representation of the influences of input files in the final product when generating a visualisation. 22
3.11 Example dynamically generated JavaScript code. 23

4.1 File tree representing all the files related to the visualisation tool 25
4.2 Example line in source code containing TraceList. 29
4.3 Example reference to a variable in the trace. 29
4.4 Error message for wrong file extensions. 30
4.5 Control bar. 31
4.6 Example of a tooltip. 31
4.7 Footer with information. 32
4.8 Control-flow-graph with dragged elements. 32
4.9 Predicate-Tree 33
4.10 Trace table. 33

List of Tables

1.1 Basic trace for Algorithm 1.1 with input: $s =$ "$aabbc$", $c =$'b' 3
1.2 Extended Trace for Algorithm 1.1 with input: $s =$ "$aabbc$", $c =$'b' 4

List of Algorithms

1.1 Example algorithm: firstOccurrence 3
1.2 Example SP code: firstOccurrence 7

3.3 Example algorithm for one of the contron-flow-graph elements that are draggable. 24

Bibliography

[And17] Andrean, Angga: *Binary Tree MLM Html.* `https://codepen.io/anggaand123/details/wexprZ`. Version: July 2017

[Bje05] Bjesse, Per: What is Formal Verification? In: *SIGDA Newsl.* 35 (2005), Dezember, Nr. 24, 1âes. `http://dx.doi.org/10.1145/1113792.1113794`. – DOI 10.1145/1113792.1113794. – ISSN 0163–5743

[BSS17] Bühler, Peter ; Schlaich, Patrick ; Sinner, Dominik: *HTML5 und CSS3: Semantik - Design - Responsive Layouts.* Springer Berlin Heidelberg, 2017. – ISBN 978–3–662–53916–3

[BSS18] Bühler, Peter ; Schlaich, Patrick ; Sinner, Dominik: *Webtechnologien: JavaScript – PHP – Datenbank.* Berlin, Heidelberg : Springer Berlin Heidelberg, 2018. – ISBN 978–3–662–54730–4

[Cha18] Chandoo, Maurice: *A Systematic Approach to Programming.* 2018

[FJJ00] Ferraiolo, Jon ; Jun, Fujisawa ; Jackson, Dean: *Scalable vector graphics (SVG) 1.0 specification.* iuniverse Bloomington, 2000

[Mar10] Marlow, Simon: *Haskell 2010 Language Report.* July 2010

[MS04] Marriott, Kim ; Stuckey, Peter: NP-Completeness of Minimal Width Unordered Tree Layout. In: *J. Graph Algorithms Appl.* 8 (2004), 01, S. 295–312. `http://dx.doi.org/10.7155/jgaa.00093`. – DOI 10.7155/jgaa.00093

[MV15] Meier, Arne ; Vollmer, Heribert: *Mathematics for application.* Bd. 4: *Komplexität von Algorithmen.* Lehmanns media, 2015. – 9,94 S. – ISBN 978–3–86541–761–9