Bachelor's Thesis

A Web Frontend for Visualization of Computation Steps and their Results in CPAchecker

Dr. Sonja Münchow

2020

SoSy-Lab LMU Munich, Germany
Supervisor: Prof. Dr. Dirk Beyer **Advisor:** Thomas Lemberger

Statement of originality

Hiermit versichere ich, dass die vorliegende Arbeit von mir selbstständig verfasst wurde und dass keine anderen als die angegebenen Quellen und Hilfsmittel benutzt wurden, sowie dass ich die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt habe.

I hereby confirm that I have written the accompanying thesis by myself, without contributions from any sources other than those cited in the text , and that I have not yet submitted the paper in the same or similar form to any other examination office.

München, Juni 2020

Dr. Sonja Münchow: _____

Abstract

CPAchecker is a highly acclaimed software verification tool, based on the CPA concepts. Configurable program analysis (CPA) allows the expression of different program analysis and model checking approaches in one single formalism. Analysis results can be viewed interactively in form of a HTML document, created by the implemented *ReportGenerator*. The report includes not only log entries, statistics, configuration properties, and source code to be verified but also two interactive graphs. CFA, which shows the control-flow automaton of the program and ARG, which shows the abstract reachability graph calculated by the analysis. Although the report is interactive, only final results of the computed and constructed abstract reachability graph are displayed. It might be sufficient to view only the results of the verification analysis, but since the CPA algorithm with its multiple analysis combining opportunities makes it reasonably challenging to follow the complex computation, a visualization of every single or at least every relevant calculation step would be an improvement of the CPAchecker.

This thesis provides a new web frontend design and implementation for visualization of computation steps and their results. The implementation includes a new CPA (CollectorCPA), acting as a wrapper of the ARG CPA. As the name suggests, it serves to collect relevant calculation steps, which are displayed in the new web frontend. The new web frontend *ComputationSteps.html* will be generated in addition to *report.html*. It includes the graphical representation of the program flow and reached abstract states in form of an interactive ARG. The final ARG can be inspected both as a standard ARG, where only the final calculation steps are displayed, and as an ARG with intermediate merging states, where the relevant states are highlighted in color. The inspection of the ARG is supported by the zoom and pan function, and the tooltip. A novelty of this implementation is the viewable interactive step-by-step construction of the ARG. The user has the possibility to go back an forth in the chronological construction of the ARG by using PREV/NEXT buttons or a slider. Source code of the C-program to be analyzed and the developing ARG or computation steps graph are displayed side by side. The source code lines are corresponding to the current edges of the ARG and are also highlighted in color. To achieve these goals we used state-of-the-art web technologies like Dagre D3 for graph generation and for the step-by-step construction of the ARG the use of D3 JavaScript libraries. The web design was realized with CSS and HTML. In summary the newly implemented CPA and web frontend help to understand the computation steps of the CPAchecker better and faster and is thus a beneficial tool for teaching and the general user.

Contents

List of Figures

1 Introduction

Motivation

Beside software testing, dataflow-analysis and model-checking are important techniques, used for the formal verification of software. CPAchecker is an open-source tool which combines the latter two approaches in just one formalism [3]. CPAchecker derived from the idea of configurable program analysis [5]. Meaning the tool can be configured to perform a variety of customized settings to fit the users focuses on either precision or efficiency. The main CPAchecker algorithm can execute a reachability analysis on freely selectable combinations of existing CPAs [3]. The CPA interface bares the opportunity for the definition of program analyses. It includes the defining of an abstract domain, which is the abstraction of concrete semantics, the defining of a transfer relation, that associates abstract states with their successors, the defining of a merge operator, which controls the merge of two states, and a stop operator, which controls whether the analysis should terminate or continue at a given state, and last but not least the defining of a precision adjustment operator for weakening or strengthening of a given abstract state.

Before a program analysis starts, the C-program to be analyzed is arranged into a control-flow automaton (CFA). The nodes of a CFA represent the locations, i.e. a program counter value, and the edges of the CFA represent program operations, for instance a function call or return. For some of the CPA configurations the program then computes an abstract reachability graph (ARG). The nodes of the ARG represent states. Those states can be told as an overestimation of the concrete reachable state. The edges of the ARG represent the transfer relation to the successor states of a reached state. Once the program analysis is finished the CPAchecker automatically generates a report of the verification run (see Figure 1.1). Both graphs, CFA and ARG, are included as graphical representations in the generated HTML-report which is located in the CPAchecker output folder. The report also contains source code of the analyzed C-program, used configuration properties, log entries and statistics. The report itself is interactive, but only final results of the computed and constructed abstract reachability graph are displayed. The user also has the possibility to view the constructed graphs, CFA and ARG, via an additional Python script and Graphviz by using the generated CFA.dot or ARG.dot file, which are also found in the output folder of CPAchecker.

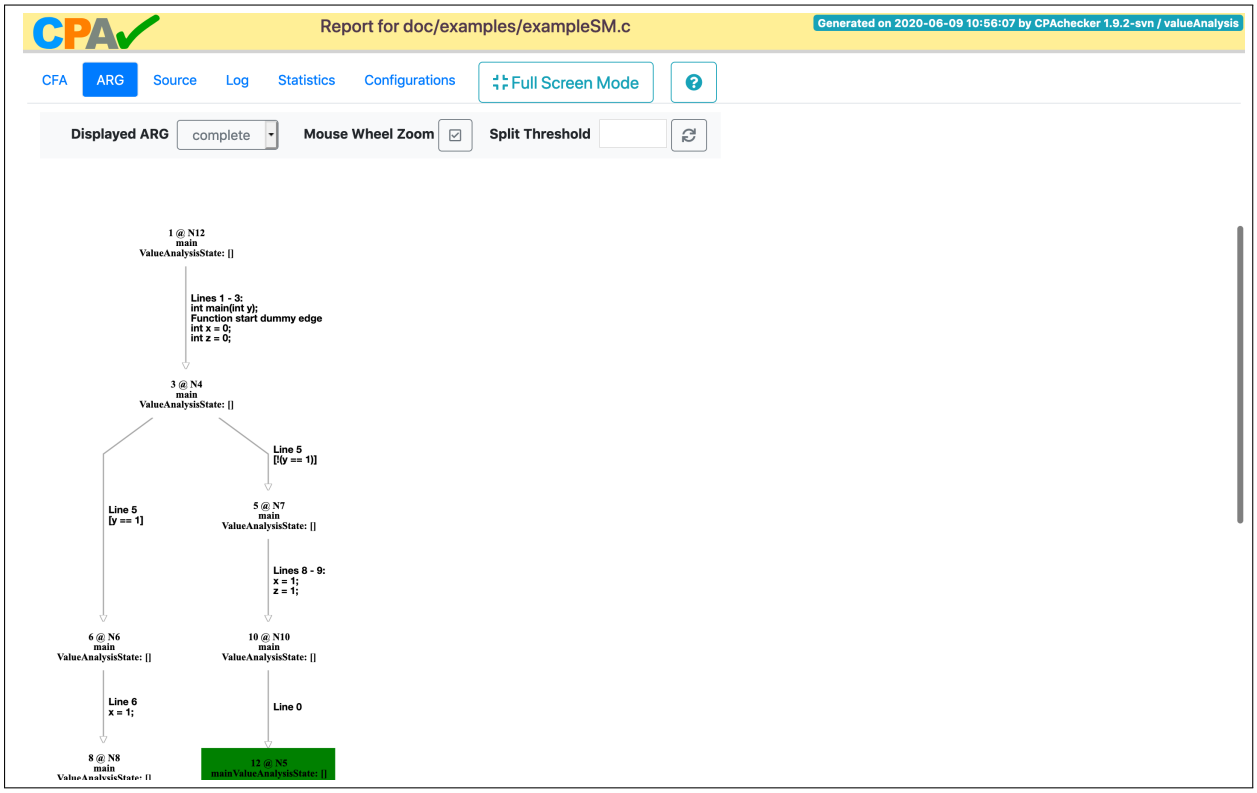


Figure 1.1: Report.html

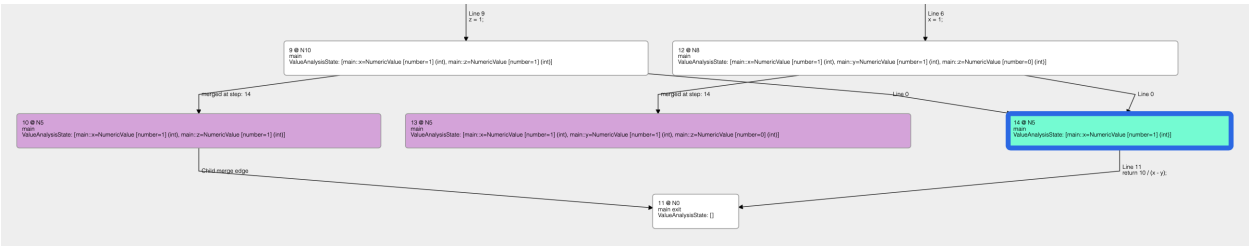
The whole report of the verification analysis gives the user a good and detailed overview of the results. However for a deeper understanding of the CPA-algorithm a visualization of computation steps is missing.

Visualization refers to the process of translating e.g. logically difficult to formulate contexts into visual media in order to make them understandable. Furthermore, visualization is used to make a certain context clear, which results from a given set of data, but which is not immediately obvious. Most people are visually oriented. It is easier to understand logically demanding contexts when you "see" them. Meaning when you see what exactly e.g. the merge operator of the CPAchecker is calculating, the whole final results becomes clearer to the one who is using the CPAchecker to verify his written C-Program. As a consequence it will become easier to find errors and inconsistencies since verification results could be faster reviewed by visual inspection.

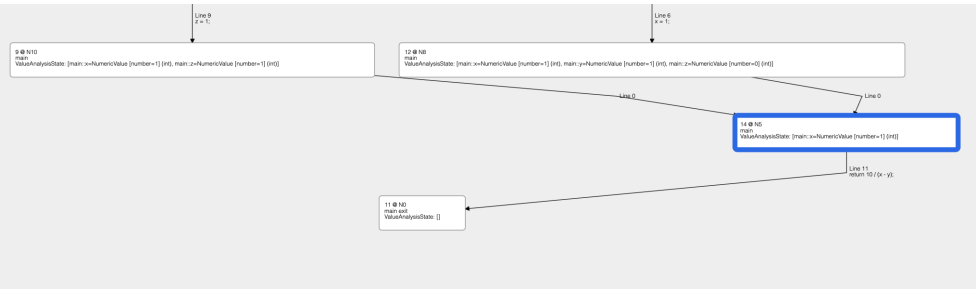
It is a typical students exercise to perform a CPA analysis manually and represent the resulting set of reachable states as an ARG. To check their manual calculated result, students can use the final ARG of the CPAchecker. As long as they made their calculations right the final ARG is adequate to validate their results, but if they made somewhere mistakes the ARG will no longer be sufficient. To follow the whole computation of reachable states,

represented by nodes in the final ARG is quite challenging. Students examining the final ARG, regarding all steps of the algorithm, often come to a point where calculation steps are difficult to understand, since the intermediate states are not displayed anymore. In the Evaluation part (5.1) of this thesis this typical problem is presented in more detail by means of a task for the survey participants.

Viewing the growth of the abstract reachability graph interactively or animated would be a feature of the CPAchecker which is not only beneficial for students, but also for the interested general user of verification software. The motivation that drove us was to visualize the construction of the already existing ARG in the report step by step instead of the final ARG view, because we hoped to make logically complex calculation steps easier to understand. One such complex calculation is the merge operation of the CPAchecker. Merged states disappear in the final ARG, only the state which is calculated by the merge operator is visible. The two states, the merge partners are no longer visible. Compare (1) and (2) in figure 1.2.



(1) Highlight Merge, showing the two merge partners in plum and the merged state in cyan



(2) Standard ARG, only the calculated merged state is visible

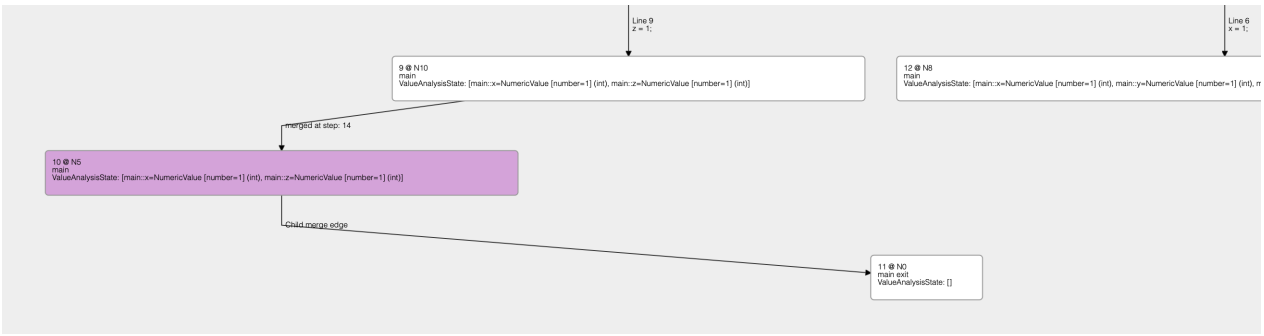
Figure 1.2: Merge Operation

Thus, information is missing in the final ARG product, which can only be calculated or guessed at with great effort. The step-by-step visualization is intended to show the exact generation and visualize the merging of states. How can this be realized? The idea we had in mind was that states appear step by step in the correct chronological order and states that are merged in the temporal future of the algorithm are somehow marked. When the merge operation is performed by the CPAchecker, it should be clearly visible which states are

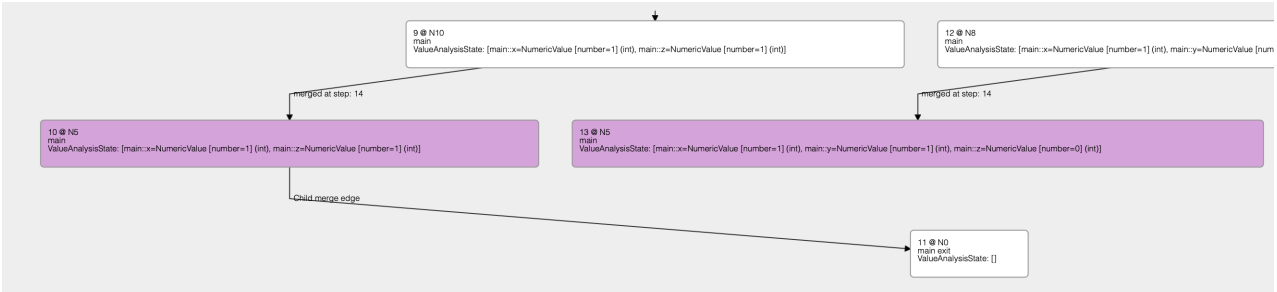
merged. The merge partners should be highlighted and appear in the correct chronological order. The first step is the calculation of the transfer relation of the first merge partner (and its successor). Followed by the appearance of the second merge partner. The next computation step of the merge operator is the merge itself with the merged element as result and the consequent disappearance of the merge partners. After the merge the ARG should show the same as the standard ARG (See Figure 1.3).

Viewing the growth of the ARG step by step could be realized either by animating the construction of the ARG or by interactively take action. Pressing PREV/NEXT buttons or operating a slider are typical interactive HTML events. Before we decided to go for an interactive web frontend, we pursued the idea using dot files for the animation, which had several disadvantages (see section 4.2). The interactive web frontend has the advantage to use interactive HTML events like PREV/NEXT buttons. The idea of going back and forth visually when creating an ARG contributes to a deeper understanding of CPAchecker's results. Compare it to a movie where you missed the key moment but have the opportunity to rewind. The result of the CPAchecker how it is shown in the final ARG is like the movie with the missed key moment. The new interactive web frontend allows you to repeat the key moment as many times as you like, respectively until you understand the key moments such as the merge operation.

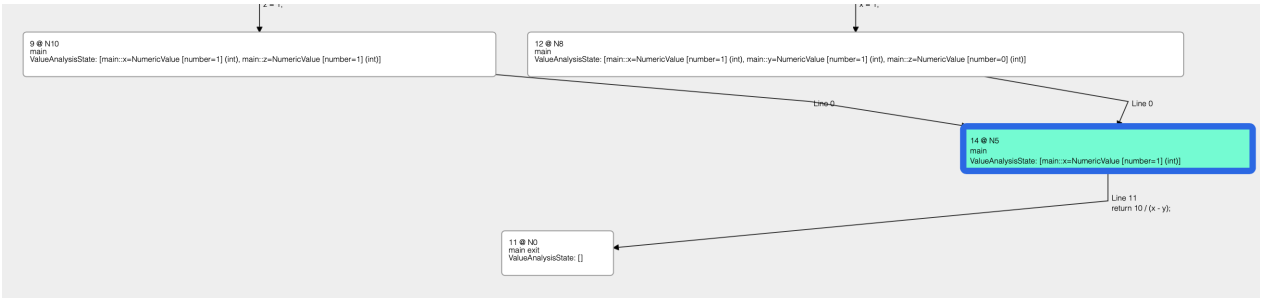
This thesis presents the new web frontend design and implementation for visualization of computation steps and their result. It required the implementation of a new CPA, *CollectorCPA* and the design and implementation of the web frontend in form of an additional HTML file, *ComputationSteps.html*. The implemented *CollectorCPA* serves as a wrapper of the already existing ARG CPA. The ARG CPA is defined in order to track reachable abstract states and build the ARG using the predecessor-successor relation of two abstract states. Wrapping the ARG CPA allows to collect and store states which will be otherwise discarded in the process of calculating reachable abstract states. Such destroyed states are for example states which result in an more abstract state by the merge operation. But those are exactly the states which will be necessary for a deeper understanding of the CPA algorithm and consequently the results of the verification analysis. The data stored by the *CollectorCPA* are passed to a JavaScript in JSON format. As for the report the new implementation also uses Dagre D3 for the graph creation using the passed data. The additional output in form of a HTML file has a new web design in order to fulfill the needs of a step-by-step construction of the ARG. In order to make the graph structure, which is actually static from Dagre D3, interactive, D3 JavaScript libraries are used. The fact that the new web front-end largely delivers what it promises was evaluated by means of a survey.



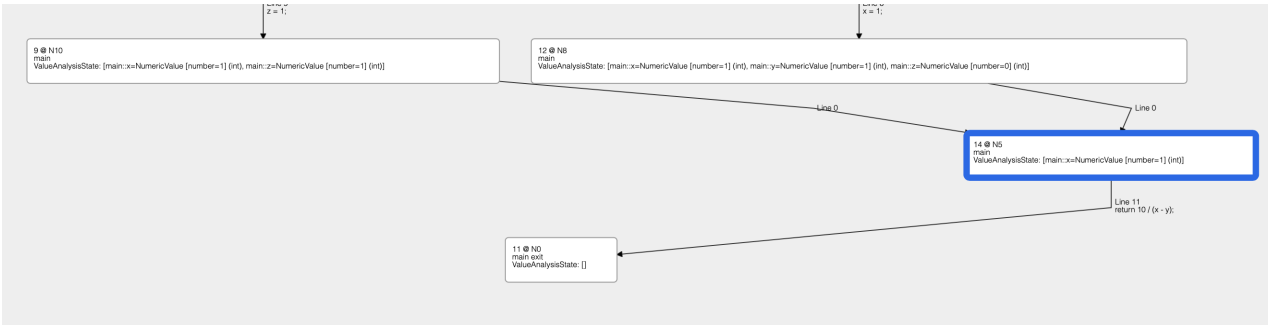
(1) First merge partner and its successor are calculated and highlighted in color



(2) Second merge partner is calculated and highlighted in color



(3) Merged element in color with passed successor



(4) Next step, the color highlighting is removed again

Figure 1.3: Step by step merge operation

2 Related Work

In this section I introduce some related approaches for visualization of procedure and results of software verification methods.

Static Verification of Linux kernel modules

Bugs in kernel modules may result in unstable operation of a kernel or an entire OS (Operating System). Zakharov et al. devised a new method for static verification of Linux kernel modules which is embedded in a configurable toolset [6]. This toolset can be used to check software written in the C programming language, since kernel modules for most of OSs are developed using C. The final step of this new method for static verification is the analysis of the verification results. One part of this analysis is that all error traces are uniformly visualized with links to the corresponding source code of the analyzed modules and the Linux kernel. To facilitate analysis of error traces the user is provided with easy navigation over error traces and over corresponding source files of modules, kernel, and contract specifications. However no error trace graphs are provided, which could further improve the analysis of the verification results of the configurable toolset.

SATVIS

Automated theorem proving is a sub-area of formal verification of software programs. A state-of-the-art theorem prover is VAMPIRE[1]. Gleis et al. designed SATVIS [1] to support interactive visualization of the saturation algorithm used in VAMPIRE. The goal was to ease the manual analysis of VAMPIRE results. SATVIS is a tool for interactively visualizing saturation-based proof attempts in first-order theorem proving. It is build on top of VAMPIRE. SATVIS visualizes the DAG (Directed Acyclic Graph)-structure of the VAMPIRE saturation as derivation graph, where a node represents a clause. Figure 2.1 shows a screenshot of the derivation graph and the interaction menu. They used pygraphviz for the graph layout and vis.js for the graph/derivation visualization. Vis.js is a JavaScript browser-based visualization library which enables manipulation and interaction with large amounts of dynamic data. In its entirety vis.js is comparable with Dagre D3. They claim that the

[1] `http://www.vprover.org/`

interactive features of SATVIS ease the task of understanding both successful and failing proof attempts in VAMPIRE and hence can be used to further development of VAMPIRE. It facilitate first-order theorem proving for both experts and non-experts.

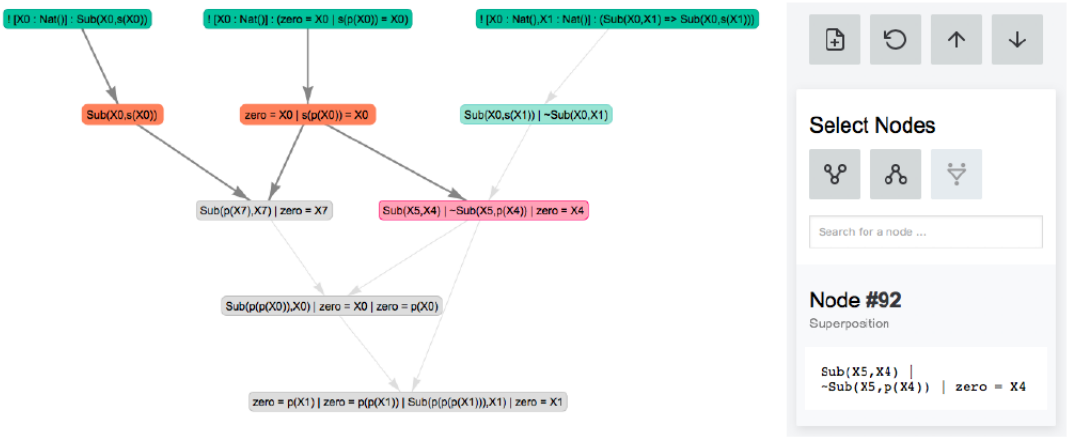
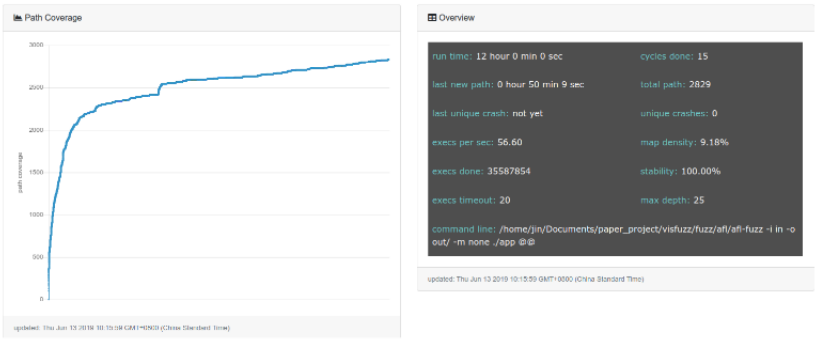


Figure 2.1: Screenshot of SATVIS, showing visualized derivation and interaction menu, taken from [1]

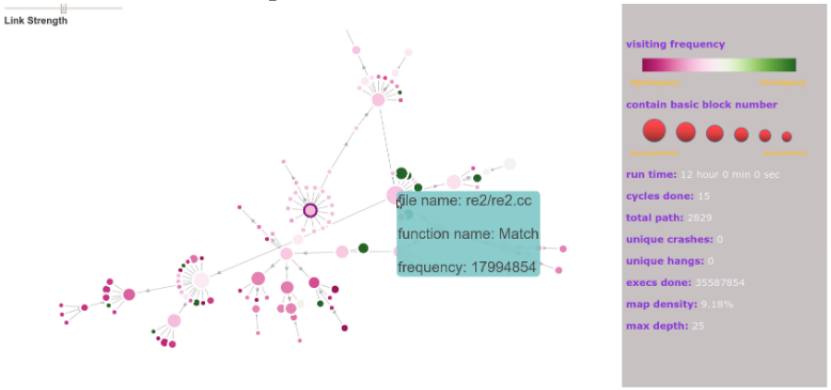
VisFuzz

Fuzzing [7] is a automated software testing technique to find implementation faults using invalid or random data as input. After inputted the invalid data, the behavior of a system is monitored to find severe bugs, like system crashes, memory leaks or unhandled exceptions. Zhou et al. [2] developed an interactive tool for better understanding and intervening fuzzing process. VisFuzz is the first tool for visualizing the fuzzing process. It is a LLVM (Low Level Virtual Machine) plugin with several features like call graphs, control-flow flow graph etc. A Python script for visualization is included. A test engineer interact with VisFuzz in 3 steps. First monitoring chart and statistics, second analyzing the call graph and the control-flow graph and last the intervention. Figure 2.2 shows the main workflow of VisFuzz. The nodes of the call graph represent functions and the edges represent the call relation between two functions. Interactive events are for example mouse hovering on a specific node which results in displaying the corresponding function. Similar to the call graph nodes, the nodes of the control-flow graph represent basic blocks and the edges represent the relation between them. The data that VisFuzz visualizes are collected static analysis results and statistics. The web application for viewing uses HTML5, Bootstrap and D3. Bootstrap is a framework for responsive web design. Since CPAchecker already uses D3 and we wanted to use it for the new web frontend as well, the D3 library is explained in section 3.2.1. In summary VisFuzz helps

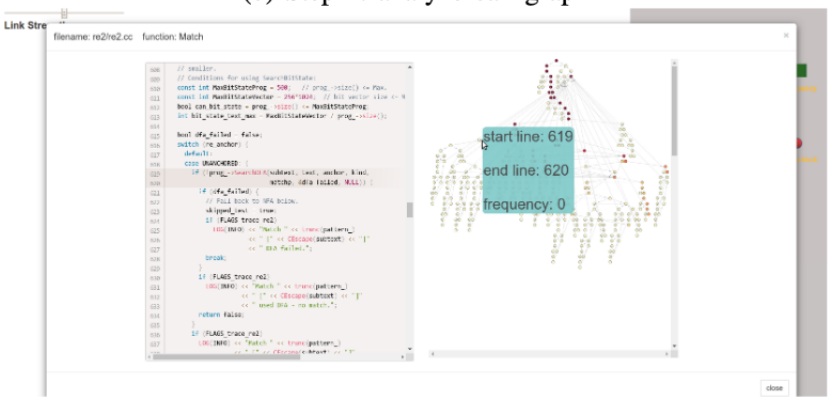
test engineers to achieve higher coverage and find more vulnerabilities in less time. This is reached because among other features of VisFuzz, the visualization helps test engineers to understand the fuzzing process.



(a) Step 1: monitor chart and statistics



(b) Step 2: analyze call graph



(c) Step 2: analyze control flow graph

Figure 2.2: Main workflow of VisFuzz, Figure taken from [2]

DIVINE

DIVINE 4 [2] is a LLVM-based model-checking simulator developed by Ročkai and Barnat. It is used for verification of C and C++ programs. The model checking analysis could be either valid or a counterexample is found. It is important to examine counterexamples interactively, since the state of a program is a very complicated structure [8]. Providing facilities for inspecting data is a main function of a simulator. A memory graph is a satisfying basis for presenting the program state to the user, since memory shortages and leaks often lead to instability of a system. Ročkai and Barnat implemented a *debug graph* by overlaying the already existing memory graph with metadata based on debug information. This enriches the nodes with more type information than the memory graph nodes, which hold only the information if a pointer exists or not. The behavior of a program as it executes, termed state space, is a time dimension. It is also visualized as a graph, where the path from a initial state to the current state can be viewed. The predecessors are the past of the program, whereas the successors correspond to possible futures of the computation. This enables to navigate back and forth in state space. When counterexamples are simulated, the stepping through the program will simply follow a counterexample. That means the user is guided through the unsound behavior of the program and stepping back and forth helps to locate the cause of the problem.

In all four examples a visualization of computation steps and results are of great benefit for the user of verification tools. A deeper understanding, even if the user has a rather low level of expertise, is achieved if the visualization is in the form of a graph. A goal, we also want to achieve by visualizing complex computation steps of the CPAchecker. SATVIS, DIVINE and VisFuzz all show an obvious advantage for the visual understanding of complex contexts when graphs are used instead of textual or tabular visualization. To show the calculation steps as a step-by-step graph should make it much easier to understand the results of the CPAchecker. In contrast to VisFuzz, however, we want to visualize not only the results but also the calculation of them in chronological order. In comparison to DIVINE, the visualization should be independent of whether a counterexample is found or not. The focus is on the visualization of the calculation steps and the results regardless of whether the verification result is true or false.

[2] `https://divine.fi.muni.cz/index.html`

3 Theoretical Background

3.1 CPAchecker

This chapter provides answers to how a CFA represents a C-program [4] (section 3.1.1), what is a CPA (section 3.1.3) and how the CPA algorithm (section 3.1.4) calculates program states on a CFA [5], how an ARG (section 3.1.2) represents those states and finally how this is all implemented in the CPAchecker [3] (section 3.1.5).

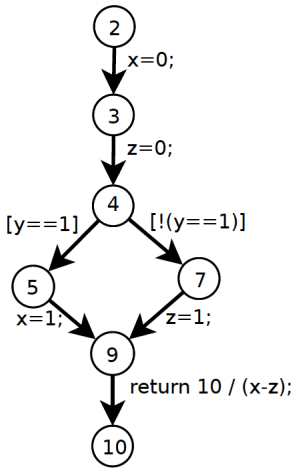
3.1.1 Control Flow Automaton

CFA

A CFA is a directed graph. The nodes of the CFA are program locations and the edges connect those locations where a transfer, namely a program operation is executed. Figure 3.1 (b) shows an example of a CFA for a corresponding C-program (a). The example is taken from [4] .

```
1  int foo(int y) {
2    int x = 0;
3    int z = 0;
4    if (y == 1) {
5      x = 1;
6    } else {
7      z = 1;
8    }
9    return 10 / (x - z);
10 }
```

(a) Example C function



(b) Control-flow automaton (CFA)

Figure 3.1: Example C function and corresponding CFA, Figure taken from [4]

A control-flow automaton $A = (L, l_0, G)$ consists of a set L of control locations, an initial location $l_0 \in L$ and a set of control-flow edges $G \subseteq L$ x Ops x L, where Ops is the set of all possible operations.

3.1.2 Abstract Reachability Graph

ARG

As a central data structure the CPAchecker constructs an abstract reachability graph (ARG). In principle, an abstract reachability graph represents an examination of the state space of a function, while making assumptions about the behavior of other functions it calls. Nodes represent the explored abstract states and edges describe the successor relation of abstract states and thus how the state space is explored.

An Abstract Reachability Graph $R = (N, i, G_{ARG})$ for a program $P = A = (L, l_0, G)$ and a given CPA $\mathbb{A}$ (see 3.1.3 ARG CPA) consists of a set $N \subseteq E_{ARG}$ of nodes (see 3.1.3 for E_{ARG}), an initial node $i \in N$, and a set $G_{ARG} \subseteq N$ x G x N of edges.

The construction of an ARG is done by continuous calculation of successors along the edges of the CFA of the C-program to be analyzed. Figure 3.2 shows an example of an ARG as you can view in the CPAchecker report. The nodes of the calculated ARG represent reachable abstract states and the edges represent program operations of the corresponding CFA. The initial node is the entry point of a program, where the analysis normally starts. If a program statement exists between two nodes respectively two abstract states, the ARG edges are labeled with this statement. If a node has no outgoing edges, it is called a final node. Final nodes represent either the exit point of a program or a target state if the ARG is incomplete. An ARG node stores the abstract state of its wrapped CPA. These abstract states include for example the formulas which represent the abstract data states, control-flow locations (program counter), and the information collected by the main CPAs used in parallel. Such used CPAs are for example CompositeCPA, LocationCPA and ValueAnalysisCPA with their own definition for the abstract domain, the transfer relation, merge and stop operator. The CompositeCPA is used for combination of multiple CPAs. For tracking the program counter explicitly the LocationCPA is used. Tracking concrete values (e.g. integers, Strings, floats or pointer) for all program variables is done by the ValueAnalysisCPA. For more details about those CPAs see [4].

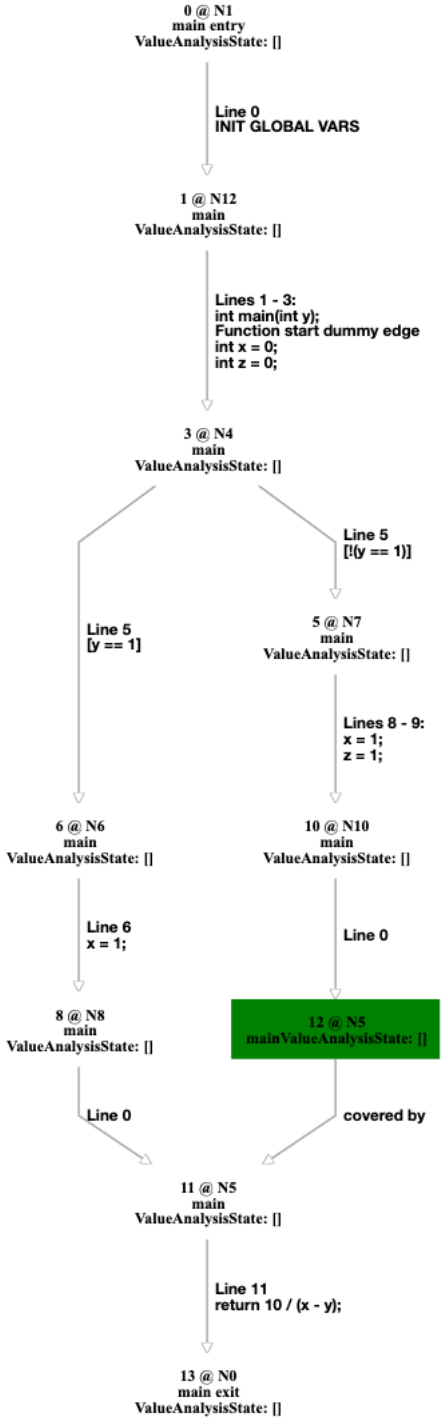


Figure 3.2: Abstract Reachability Graph

3.1.3 Configurable Program Analysis

Concrete state

A concrete state c is a variable assignment that assigns to each variable in $X \cup \{pc\}$ a value. The set X is the set of all program variables and the program counter pc is the representative of the program location.

CPA

A CPA $\mathbb{D} = (D, \Pi, \rightsquigarrow, \mathsf{merge}, \mathsf{stop}, \mathsf{prec})$ consists of an abstract domain D, a set of precisions Π, a transfer relation $\rightsquigarrow$, a merge operator merge, a termination check stop, and a precision adjustment prec [5].

The abstract domain $D = (C, \varepsilon, [\![\cdot]\!])$ is defined by the set C of concrete states, a semi-lattice $\varepsilon = (E, \sqsubseteq, \sqcup, \top, \bot)$ and a concretization function $[\![\cdot]\!]$. A semi-lattice consists of a set E of elements, a partial order $\sqsubseteq$, a join operator $\sqcup$, and a top element $\top$ and bottom element $\bot$. The elements from E are abstract states. The concretization function $[\![\cdot]\!] : E \to 2^C$ assigns to each abstract state the set of concrete states that it represents.

The transfer relation $\rightsquigarrow \subseteq E$ x G x E assigns to each abstract state e possible new abstract states e' that are abstract successors of e, and each transfer is labeled with a control-flow edge g.

For soundness and progress of the program analysis the abstract domain and the transfer relation have to satisfy several requirements.

The top element of abstract states has to represent all possible concrete states whereas the bottom element has to represent none. $[\![\top]\!] = C$ and $[\![\bot]\!] = \emptyset$

The join operator has to be precise or over-approximating, meaning a join of two abstract states has to be the same or a more concrete state that the union of two concrete states. $\forall e, e' \in E : [\![e \sqcup e']\!] \supseteq [\![e]\!] \cup [\![e']\!]$.

If an abstract state e is smaller than another abstract state e' the represented concrete states must be a subset of the concrete states e' is representing. $\forall e, e' \in E : e \sqsubseteq e' \Rightarrow [\![e]\!] \subseteq [\![e']\!]$.

The transfer relation has to be total and over-approximating in regards to program operations. $\forall e \in E : \exists e' \in E : e \rightsquigarrow e'$ and $\forall e \in E, g \in G : \bigcup_{e \overset{g}{\rightsquigarrow} e'} [\![e']\!] \supseteq \bigcup_{c \in [\![e]\!]} \{c' | c \xrightarrow{g} c'\}$. The union of all concrete states represented by all possible abstract successors of e and a program statement g need to be the same or more than the union of all concrete successors of g and all concrete states represented by e.

The merge operator merge: E x $E \rightarrow E$ combines the information of two abstract states. The result of $\mathsf{merge}(e, e')$ is an abstract state and can be anything between e' and $\top$. That means the merge operator weakens the second parameter depending on the first parameter. The resulting state can hence only be more abstract. The merge operator is defined either in $\mathsf{merge}^{sep}(e, e') = e'$ or $\mathsf{merge}^{join}(e, e') = e \sqcup e'$.

The stop operator stop: E x $2^E \rightarrow \mathbb{B}$ checks if the abstract state that is given as first parameter is covered by the set of abstract states given as second parameter. To ensure soundness of the termination check, the value $\mathsf{stop}(e, R) = true$ has to imply $[\![e]\!] \subseteq \cup_{e' \in R}[\![e']\!]$. If the value is *true* the CPA-Algorithm skips further analyzing the successor states of e.

Each of the configurable components of a CPA influences precision and cost. The precision adjustment component can either strengthen or weaken an abstract state. It is defined as $\mathsf{prec} : E$ x Π x $2^{E \text{x} \Pi} \rightarrow E$ x Π.

ARG CPA

The ARG CPA is defined in order to track the abstract states in the *reached* set of the CPA-Algorithm, and build an abstract reachability graph with the stored predecessor-successor relationship between two abstract states. Using both, the ARG CPA and the LocationCPA allows to reconstruct from an abstract path the path represented in the CFA. An abstract path is a *sequence* $\langle e_0, ..., e_n \rangle$ of abstract states such that for any pair (e_i, e_{i+1}) and $i \in \{0, ..., n-1\}$ either e_{i+1} is an abstract successor of e_i or the result of merge [9].

Each node of the ARG is labeled with a unique ID and an abstract state. A labeled node is denoted as $n : (i, a)$, where n is the node, i a unique *Integer*, and a an abstract state. Each edge of the ARG is marked with a basic block, an assume predicate, a function call or a return.

An ARG CPA $\mathbb{A} = (D_{ARG}, \Pi_{ARG} \rightsquigarrow_{ARG}, \mathsf{merge}_{ARG}, \mathsf{stop}_{ARG}, \mathsf{prec}_{ARG})$ consists of an abstract domain D_{ARG}, a set of precisions Π_ARG, a transfer relation $\rightsquigarrow_{ARG}$, a merge operator merge_{ARG}, a termination check stop_{ARG}, and a precision adjustment prec_{ARG}.

The abstract domain $D_{ARG} = (C_{ARG}, \varepsilon_{ARG}, [\![\cdot]\!])$ is defined by the set C_{ARG} of concrete states, a semi-lattice $\varepsilon_{ARG} = (E_{ARG}, \sqsubseteq, \sqcup, \top, \bot)$ and a concretization function $[\![\cdot]\!]$. The elements from E_{ARG} are abstract states. The concretization function $[\![\cdot]\!] : E_{ARG} \rightarrow 2^{C_{ARG}}$ assigns to each abstract state the set of concrete states that it represents.

The transfer relation $\rightsquigarrow_{ARG} \subseteq E_{ARG}$ x G x E_{ARG} assigns to each abstract state e possible new abstract states e' that are abstract successors of e, and each transfer is labeled with a control-flow edge g.

The ARG merge operator is defined in $\mathsf{merge}^{sep}(e, e') = e'$.

The termination check stop checks if the given abstract state with the given precision is covered by the set of abstract states given as second parameter.

The precision adjustment computes a new abstract state and precision for a given abstract state, a given precision and a given set of abstract states with precision.

CollectorCPA

At the end of a verification run the report visualize the final Abstract reachability graph. The ARG is part of the calculated reached set. The states which result in new merged abstract states are removed from the final set of reached states. Our goal was to visualize computation steps of CPAchecker like the merge operation. Since destroyed or removed states after a merge are missing in the final set of reached states it is important to store those states. But how can you store states from other CPAs without affecting the respective CPAs used?

This is done by building a wrapper CPA around the ARG CPA. Since the function of this wrapper CPA is to collect and store states that would have been lost after a merge, we have called it CollectorCPA. The CollectorCPA $\mathbb{D}_c = (D_c, \Pi_c \rightsquigarrow_c, \mathsf{merge}_c, \mathsf{stop}_c, \mathsf{prec}_c)$ consist of a collector abstract domain, a set of precisions, a collector transfer relation, a collector merge operator, a collector stop operator, and a collector precision adjustment.

The abstract domain $D_c = (C, \varepsilon_c, [\![\cdot]\!])$ is defined by the set C of concrete states, a semi-lattice $\varepsilon_c = (E_c, \sqsubseteq, \sqcup, \top, \bot)$ and a concretization function $[\![\cdot]\!]$. The elements from E_c are abstract states. Each element e in E_c is defined as tuple (a_c, H_{merge}) where $a_c \in E_{ARG}$ is the current *ARGState* and H_{merge} a tuple (a_c, a'_c, a''_c), namely the recorded merge operation. Each tuple (a_c, a'_c, a''_c) describes the merge operation $merge(a'_c, a''_c) = a_c$ with $a_c, a'_c, a''_c \in E_{ARG}$. The concretization function $[\![\cdot]\!] : E_c \to 2^C$ assigns to each abstract state the set of concrete states that it represents.

Even though the collector state is able to store additional information, all the analysis operations are applied on the wrapped states which are ARG states [Figure 3.3].

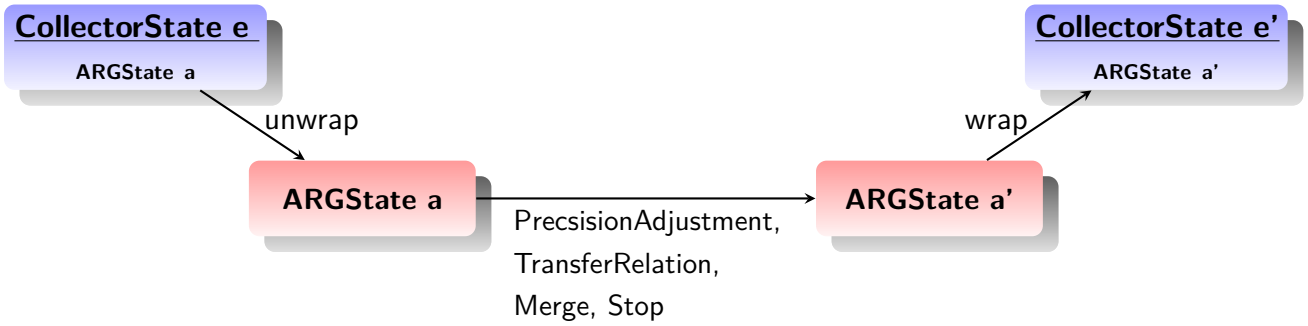


Figure 3.3: Unwrapping of CollectorState for CPA operations

The transfer relation $\rightsquigarrow_c \subseteq E_c$ x G x E_c assigns to each abstract state e possible new abstract states e' that are abstract successors of e, and each transfer is labeled with a control-flow edge g. This is done by unwrapping each abstract state e and subsequent calculation of possible successors of $e_{unwrapped}$ which then will be wrapped and assigned as new abstract states e'. For $e_{unwrapped} = a_c$ the transfer relation $\rightsquigarrow_{ARG} \subseteq E_{ARG}$ x G x E_{ARG} assigns to each abstract state a_c possible new abstract states a'_c that are abstract successors of a_c, and each transfer is labeled with a control-flow edge g. The transfer relation has to be total. $\forall e \in E_C : \exists e' \in E_C : e \rightsquigarrow e'$ if it is valid that $\forall a \in E_{ARG} : \exists a' \in E_{ARG} : a \rightsquigarrow a'$. We write for a given $e = (a_c, H_{merge}) \rightsquigarrow (a'_c, H_{\{\}}) = e'$ if for $a_c \in E_{ARG} : \exists a'_c \in E_{ARG} : a_c \rightsquigarrow a'_c$. For each e' the tuple $H_{\{\}}$ is an empty set.

The merge operator is defined as delegated merge
$\mathsf{merge_c}((a'_c, H'_{merge}), (a''_c, H''_{merge})) = (\mathsf{merge_{ARG}}(a'_c, a''_c), H_{merge}) = (a_c, H_{merge})$
As additional information to the merged element the new merged collector state will store a' and a'' as $H_{merge} = (a_c, a'_c, a''_c)$ where a_c is the result of $\mathsf{merge_{ARG}}(a'_c, a''_c)$. The merge history in form of H'_{merge} and H''_{merge} will not be passed to the new merged collector state since this information is irrelevant for the visualization of a current merge operation. For a short overview see Figure 3.4. The figure illustrates how two *CollectorStates* are merged into a new *CollectorState*. As shown in Figure 3.3 the elements must be unwrapped for the merge operation. More details are discussed in the implementation part of this thesis.

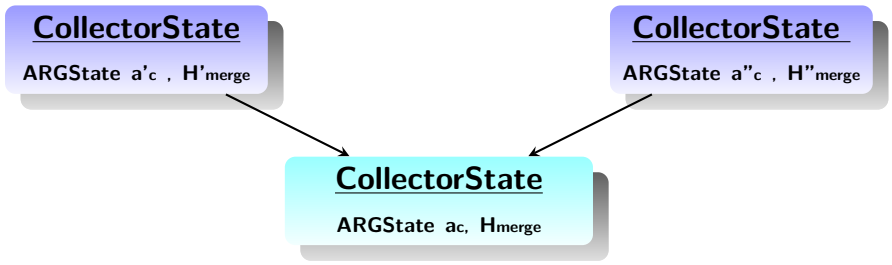


Figure 3.4: Merge of two CollectorStates

The termination check stop^{sep} considers each abstract state individually. If the check returns $\mathsf{stop}^{sep}(e, R) = true$ because it is already covered the analysis of further successors of e is omitted.

The precision adjustment is also delegated to the wrapped state. If the CPAchecker computes a new precision the analysis is continued with this precision adjustment otherwise nothing is changed.

3.1.4 CPA-Algorithm

For a given CPA and an initial abstract state (see Input in Figure 3.5) the reachability algorithm computes a set of reachable abstract states. This set is an over-approximation of the set of reachable concrete states. The set *reached* stores all reachable abstract states already visited, and the set *waitlist* stores all abstract states yet not processed. Both sets are being updated during the execution of the CPA-algorithm. It starts with the initial abstract state e_0. In the beginning the initial state is the only state included in both sets. As long the *waitlist* set is not empty, a state from *waitlist* is chosen as current state and each possible successor obtained from the transfer relation is examined. Every abstract successor state is then combined with an existing abstract state from the *reached* set using the given merge operator. In case that the merge operator does not generate a new combined state, it simply returns the existing abstract state that was given as second parameter to the merge operator. Otherwise if the merge operator has added information to the new abstract state, such that the old abstract state is included, then the old abstract state is replaced by the new one. The existing state is removed from *waitlist* and *reached* and the new one is added instead. After the current successor state has been merged with all existing abstract states, the stop operator determines whether needs to store the current state in *reached* and *waitlist*. The iteration stops when the *waitlist* set is empty and the *reached* set will be returned. It contains all reached and analyzed states. The CPAchecker uses then an ARG as model for visualization.

Algorithm 1: ***CPA***$(\mathbb{D}, e_0)$

Input : a CPA $\mathbb{D} = (D, \rightsquigarrow, \mathsf{merge}, \mathsf{stop})$, an initial abstract state $e_0 \in E$ where E denotes the set of elements of the lattice of D

Output : a set of reachable abstract states

Variables: a set reached $\subseteq E$, a set waitlist $\subseteq E$

```
1  waitlist := {e_0};
2  reached := {e_0};
3  while waitlist ≠ {} do
4      choose e from waitlist;
5      waitlist := waitlist\{e};
6      for each e' with e ⇝ e' do
7          for each e'' ∈ reached do
8              //combine with existing abstract state;
9              e_new := merge(e', e'') //Merge Operator ;
10             if e_new ≠ e'' then
11                 waitlist := (waitlist ∪ {e_new})\{e''};
12                 reached := (reached ∪ {e_new})\{e''};
13             end
14         end
15         if ¬stop(e', reached) //Stop Operator then
16             waitlist := waitlist ∪ {e'};
17             reached := reached ∪ {e'};
18         end
19     end
20 end
21 return reached //Output
```

Figure 3.5: CPA-Algorithm [4]

3.1.5 CPAchecker

CPAchecker[1] is a software verification tool which unifies model checking and program analysis. Data-flow analysis and model checking are two approaches in software verification to prove the correctness of a program according to a given specification. The given specification include liveness (something good eventually happens) and safety (nothing bad happens) properties. The information you get by data-flow analysis or model checking are used to validate safety properties [4]. One approach for this is called reachability analysis, which checks

[1] `https://cpachecker.sosy-lab.org/`

if an execution path of a program exists that reaches a specific location in the program. A program is represented by a control-flow automaton (CFA). The CPAchecker analyzes the program on this intermediate representation of the program using specific CPAs (Configurable Program Analyses). To visualize the flow of a program, an abstract reachability graph (ARG) will be constructed. The reachability tree contains all reachable abstract states according to the transfer relation [4]. The construction of an ARG is done by traversing the CFA and calculating successor states using the information of the edges (program operations) of the CFA.

The CPA algorithm is the core of CPAchecker. Since it applies all operations on an abstract data type, the concrete analyzing CPA is not necessary to know. For most configurations the composite CPA is used. It allows the combination of multiple different CPAs. In general the ARG CPA is the most outer CPA, so that an abstract reachability graph is available at the end of a verification run.

Figure 3.6 [3] shows the interaction of CFA, main CPA and additional CPAs. The CFA represents the C-program to be analyzed. CPAchecker has all required interfaces and operations for the implementation of a new user-defined CPA. It is possible to implement a new CPA as CompositeCPA if a combination of different CPAs is desired or as a Leaf CPA for stand-alone usage or as part of compositeCPA.

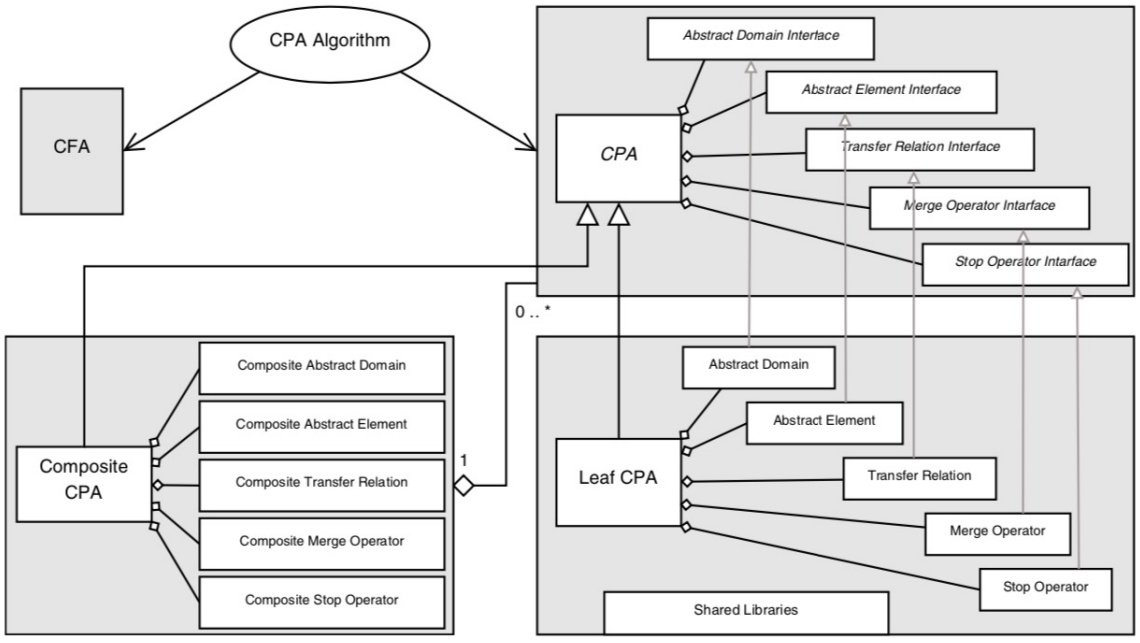


Figure 3.6: Design for implementation, Figure taken from [3]

3.2 Used Libraries and scripts

3.2.1 D3

D3[2], short for Data-Driven documents is a JavaScript library for interactive visualization of data in web browsers. D3 provides prebuilt functions for creating and manipulating SVG (Scalable Vector Graphics) objects. Large data sets can be bound to the SVG objects and you have the possibility to style them using CSS (Cascading Style Sheets). The prebuilt functions that allow you the manipulation of a DOM (Document Object Model) include selections, transitions, array operations and a variety of maths functions. There are many functions for 2D transformations, like translation, scaling, and rotation. The data you would like to bind and process with D3 can be in different formats e.g. in JSON (JavaScript Object Notation) format. CPAchecker already uses JSON , which are easy human-readable objects consisting attribute-value pairs and array-data types. The D3 library also provides multiple pre-built functions that enables the user for array manipulation, like removing and sorting array elements. Together with selection you can find specific elements and manipulate them individually. Elements can be selected in different ways, either by an attribute, a HTML tag, a unique identifier or a CSS class. Once an element is selected you can modify the associated properties (like shape, colors and values) and behaviors (like transitions and events). On the D3 website[3] you will find a detailed online documentation with many examples and tutorials. Since D3 with all its prebuilt functions for data manipulating and visualization is right between a data processing and pure graphics library, D3 fulfills all the data visualization requirements we need for the CPAchecker. Even though it would have been possible to draw directed graphs with D3, another library (Dagre D3, Section 3.2.2) facilitates this challenge.

3.2.2 Dagre D3

Dagre D3[4] is a JavaScript library especially for lay out directed graphs. It is a D3-based renderer for dagre[5]. The layout is completely client-side computed. That means the technical environment of the user can be taken into account. The processing is done directly from the browser. The request is redirected to a HTML file without content. Once the layout information (Data and JavaScript) will be supplied, the browser compiles everything before rendering the content. On the basis of faster processing at the expense of security, Dagre needs only rudimentary information to lay out medium-sized graphs quickly. For the cre-

[2] `https://d3js.org/`
[3] `https://d3js.org/`
[4] `https://github.com/dagrejs/dagre-d3`
[5] `https://github.com/dagrejs/dagre/wiki`

ation of a graph dagre uses the graphlib API[6]. Graphlib has only one graph type which is per default directed. All functions necessary for nodes need user-supplied String ids for unique identification. Edges are identified by nodes they connect. Dagre uses several different algorithms for drawing nice graphs [10] with minimum edge crossing and perfect matching coordinates for the nodes [11]. For a more detailed description of the used algorithms have a closer look at the papers recommended on the dagre wiki site. Although it is an advantage not to have to calculate the best node and edge positions yourself, Dagre D3 also has disadvantages. Beside some minor issues, like less well documentation, a major disadvantage of using Dagre D3 is the fact, that a stepwise creation of graphs is not supported.

3.2.3 jQuery

The jQuery library greatly simplifies programming using the popular language JavaScript. Features of jQuery are for example HTML/DOM manipulations, CSS manipulations, effects, and HTML event methods. The basic syntax of jquery is: $(selector).action()[7]. With the $ sign you access jQuery, then you select your HTML element and manipulate it. JQuery selectors are based on CSS selectors. It is possible to select by id, name, classes and so on. You can combine different selectors for a more precise finding of an element. The methods which take action on elements include changing values of attributes, and adding and removing CSS classes . Additionally jQuery provides several methods to handle events. For example the command $(#button).click() triggers the mouse click event when the id of the selected element matches "button". For defining "what should happen" when the event is activated it is necessary to pass a function to the event. The function can be anything from a single statement to a block of multiple commands to perform the task you would like to bind to an event. For further information the jQuery API documentation gives a detailed overview on jquery featured methods[8].

3.2.4 Dot_to_ gif_sh

In addition to the *Report.html* file CPAchecker generates several DOT files, including CFA and ARG DOT files. Those files harbour all graph-relevant information for nodes and edges and their attributes as Strings. DOT is a plain-text graph description language. The DOT syntax describes or defines a graph but has no capability for rendering a graph. To view a graph in DOT language, an additional program is needed. Graphviz is such a graph visualization software [9]. The graph description in its simple text language is used

[6] `https://github.com/dagrejs/graphlib/wiki/API-Reference`
[7] `https://www.w3schools.com/jquery/default.asp`
[8] `https://api.jquery.com/`
[9] `www.graphviz.org/`

by Graphviz to create SVGs (Scalable Vector Graphics) for web pages. Note that an older version of CPAchecker used only those DOT.files for graph visualization. It was necessary to run an external Python script which passed the DOT files to the Graphviz library [11].

As a first approach we wanted an animation of the ARG construction for visualization of computation steps of the CPAchecker. GIF (Graphics Interchange Format) is an image format, which can be used for small animations. A series of frames is be combined to produce this sort of animated GIFs. Dot_to_ gif_sh [10] is an easy and short shell script for creating a GIF from DOT files using Graphviz. It enables to use multiple DOT files, each file representing one step of a graph-building algorithm to create an animated GIF. Once you have all your separate DOT.files representing each a snapshot of the graph building in one directory you execute the shellscript using for example the command ./dot_to_gif.sh $(find dot -name '*.dot'). That means all files in your directory ending with .dot will be used for creating animation.gif. The command *dot* tells Graphviz to draw directed graphs. The animation.gif can then be viewed in a web browser.

3.3 CPAchecker report

The CPAchecker report is an automatically generated HTML file. To provide SoC (Separation of Concerns) CPAchecker combines three template files, namely HTML, CSS, and JS to one single HTML file. The report.html file is found in the output folder inside the CPAchecker directory. The user can view the report.html in any arbitrary web browser. In case the result of the verification run is that the C-program is not proven correct the report is named different. The user finds out the exact name of the report at the end of the command-line output of the CPAchecker. Even though the DOT files of an older version of CPAchecker are still generated, and delivered in the output folder, there are no longer required for building graphs (e.g. ARG and CFA) in report.html. The latest version of CPAchecker uses D3 and Dagre D3 for rendering the graphs. The data required for graph creation are given to the report after the verification run is terminated. As already mentioned the data is provided in JSON format. JSON is a language-independent data format. It is human-readable and the basic data types are Number (no distinction between integer and floating-point), String, Boolean, Array, Objects (unordered collection of name-value pairs), and null.

The data which is necessary for the construction of the abstract reachability graph is supplied as a quite simple structured JSON object. It consists two keys, *nodes* and *edges*. The keys contain arrays of objects, which hold all the information, important for the nodes and edges of the ARG. Figure 3.7 shows an example of how such an array of objects looks like.

[10] `https://gist.github.com/maelvls/5379127`

Each node object has the information about the function in which the node is present, its index, which represents the ARG ID, a label and the type of the node. Each edge object carries information about the file in which the edge is contained, the source-code line, the source, which is the node index where the edge starts, a label and its target, which is the index of the node, the edge is pointing to.

```
{"nodes": [
  ...,{...},
  {"func":"main",
  "index": 42,
  "label":"40 @ N10\nmain\nValueAnalysisState: [main::a=
   NumericValue [number=3] (int), main::i=NumericValue [number
   =3] (int)]\\",
  "type":""}, {...},...],
"edges": [
  ...,{...},
  {"file":"doc/examples/example.c",
  "line":"13",
  "source":42,
  "label":"Line 13\n[!(i != a)]",
  "type":"AssumeEdge",
  "target": 43},
  {...}, ...]
}
```

Figure 3.7: Structure of JSON object for ARG

Once the verification run is terminated, the HTML template is read line by line, and as soon as the current line contains a predefined string, this string will trigger a specific action. It is the *ReportGenerator.java* class which possesses methods for certain actions. The action-causing strings are included in the HTML template as *meta* tags. The *meta* tag $<!--\mathsf{REPORT\_CSS}-->$ causes the ReportGenerator to read the CSS template, and the *meta* tag $<!--\mathsf{REPORT\_JS}-->$ to read the JS template and insert them line by line into the HTML template. There are other *meta* tags which trigger insertion of dynamic data like statistics, configuration etc. The JavaScript template contains also action-causing strings. For example reading ARG_ JSON_ INPUT causes the ReportGenerator to write the JSON data required for the construction of the ARG and insert it into the JavaScript

template respectively into the HTML file as *var argJson*. The HTML template only lists the content, the JavaScript template processes the inserted data (behavior) and the CSS template contains the style information. All three assembled in one file enables an easy transfer of the verification results between users.

Both graphs, CFA and ARG are constructed using the JSON data the CPAchecker produces. An empty Dagre D3 graph will be populated with nodes and edges. The essential information for the nodes and edges are provided in the appropriate JSON object and processed by the JavaScript using the Dagre D3 library. The resulting graphs can then be viewed by the user. An online tutorial about the features of the CPAchecker report is available[11].

[11] `https://sosy-lab.gitlab.io/research/tutorials/CPAchecker/HTMLReport.html`

4 Implementation

This chapter will focus on the implementation of the new wrapper CPA alias CollectorCPA in section 4.1 (for the definition see page 22 ff.), the collection and preparation of the relevant graph data in section 4.4 and the integration of the new web frontend into the CPAchecker in section 4.5. In addition it will give a short outline on a first approach using DOT files and why this approach was not pursued further (section 4.2).

4.1 CollectorCPA

In order to extend the CPAchecker by integrating a new CPA two steps are mandatory. First an entry in the global properties file and second the implementation of the new CPA with all operation interfaces [3].

The collector.properties file stores the collector configuration. It defines the collectorCPA as wrapper of the ARG CPA. It tells the main CPA that the concrete CPA is the CollectorCPA and that this CPA should use the ARG CPA and the CompositeCPA, which in turn defines that the LocationCPA, ValueAnalysis, and CallstackCPA are combined.

The configuration of CPAchecker is done via configuration files and command-line arguments. The configuration file specify a set of options and the argument -setprop <KEY>=<VALUE> sets any option: KEY = VALUE. The file doc/ConfigurationOptions.txt in CPAchecker contains an explanation of these options and command line switches are described in doc/Configuration.md. To run the analysis you need to use the following arguments:
-setprop solver.solver=smtinterpol
-setprop cpa.predicate.encodeBitvectorAs=integer
-setprop cpa.predicate.encodeFloatAs=rational
-collector doc/examples/exampleSM.c
The arguments -config config/CONFIGFILE.properties can be abbreviated to -CONFIGFILE, meaning using -collector is sufficient. If you would like to use another C-program example, than exampleSM.c you need to preprocess your program with the C pre-processor.

The implementation of the Wrapper CPA, we named CollectorCPA follows the principles of implementing a Composite CPA with only one associated CPA, the wrapped ARG CPA. The

CollectorCPA returns for all CPA Operations the corresponding Collector operations. The newly implemented collector operations all unwrap the abstract collector states and execute the operations on the unwrapped ARG state by using the ARG operations. Subsequently the information about the current state and the computated state (e.g. successor, merged element) is stored as *ARGStateView* (Section 4.3) representing the original *ARGState,* followed by wrapping it again as new *CollectorState*.

The *TransferRelation* as well as *PrecisionAdjustment* and the termination check operator *Stop* are implemented according the formalism described in subsection 3.1.3.

An existing *ARGState* could be getting destroyed for example, after a merge occurred. If an *ARGState* is destroyed, the only information which remains is the state ID of this *ARGState*. The final abstract reachability graph does not need any information about states, which will be merged. The essential information about predecessors (parents) and successors (children) will be passed to the merged element. In order to build an abstract reachability graph step by step, taking into account each calculated merge step, we have to remember states that will be merged. The implemented *CollectorMerge* Operator stores the merge partners, the states which will be merged, before it delegates the merge operation to the ARG merge operator. To avoid dynamic value changes of *ARGStates* caused by the fact that Java uses *pass by value*, we need to store the states as *ARGStateView* (Section 4.3). After the unwrapped original *ARGStates* are passed to the ARG merge Operator and the merged *ARGState* element is returned, the merged element is saved additionally as *ARGStateView*. When rewrapping the new *CollectorState* all three *ARGStateViews*, the two elements which will be merged and the merged element, are passed.

The CPA Algorithm collects reachable states. At the end of a verification run, the collection of reachable states is passed as *UnmodifiableReachedSet* to the method *printStatistics* in the class *CollectorStatistics*. The structure of *CollectorStatistics* enables to print additional statistic information into the default statistic text file, the CPAchecker creates after the analysis stopped. But the main functionality of *CollectorStatistics* is to create the additional HTML file *ComputationSteps.html*. The preliminary purpose of *CollectorStatistics* was to create DOT.files instead of the HTML file. The detailed approach is described in section 4.2. For both approaches (HTML and DOT) the implementation included the preparation and processing of graph-relevant information out of the proven as reachable *CollectorStates* collected in the *UnmodifiableReachedSet*. For the detailed description of graph data processing see the section 4.4 Graph Data.

4.2 DOT Graph

The first visualization approach we were pursuing was the idea of an animated graph using DOT files. This approach is available as a preliminary version in CPAchecker branch SonjaM revision 31311 [1], [2].

The CPAchecker has already existing code (*ARGToDotWriter.java)* to write DOT files with information about nodes and edges, required for an abstract reachability graph. Reusing this code assumes that the abstract states, which represent the nodes are *ARGStates*. So the first challenge was "How do we get *ARGStates* which are not in the reached set and potentially destroyed?"

As already mentioned an *ARGState* gets destroyed after a merge occurred, but this is exactly the computation step we want to visualize. By saving the information about merge partners, their parents and their children in the *CollectorState* of an merged element, the required information of how the merge was calculated is saved. However the information is not saved as *ARGState* but as *ARGStateView* (Section 4.3), which is necessary due to the fact that the *ARGState* values could dynamically be changed during the verification run. Two solutions are conceivable, first the implementation of a *CollectorToDotWriter* or second the reconstruction of *ARGStates*. Since we already have exactly the same information in *ARGStateView* as in the corresponding *ARGState* we went for the second solution.

The reconstruction of the *ARGStates* starts with extracting the wrapped *AbstractState* from the first element of the *UnmodifiableReachSet* and assign a new *ARGState*. For further requests it is necessary to link each new *ARGState* with the current *ARGState* of each iteration entry in a Hashmap. We iterate over each entry of the collection of reached states. In case the entry was calculated in *TransferRelation* we take the current wrapped*AbstractState* out of the current entry and the parent by taking the corresponding *ARGState* out of the *linkedHashMap*. Then a new *ARGState* is assigned by using the current wrapped *AbstractState* and the linked parent *ARGState*. In case *entry* is a calculated merged element, the merged element is processed equally, and in addition we need to reconstruct the saved merge partners by passing likewise the current wrapped *AbstractState* and the linked parent *ARGState*.

In order to achieve single DOT files, each representing a step of the evolution of the ARG, every newly reconstructed *ARGState* is put into a collection. The elements of this collection represent the reachable states, respectively the nodes of the ARG. Every time after adding a new *ARGState* to the collection, a new DOT file is generated. So each DOT file provides the information about the nodes and edges from start until the last added element. The files are numbered in ascending order and stored in a directory created for this purpose. The user

[1] `https://github.com/sosy-lab/cpachecker/commits/SonjaM`
[2] `https://svn.sosy-lab.org/software/cpachecker/branches/SonjaM/?p=31311`

will find the *CollectorDotFiles* folder in the general output folder of the CPAchecker.

The user needs to download the shellscript dot_ to _ gif _ sh [3]. This shellscript is used as raw template we wanted to customize. The DOT files and script need to be in the same directory. To run the script use the console command *./dot_to_gif.sh $(find dot -name '*.dot' | sort -n -t_ -k2)*. The command *sort -n* will sort the DOT files numeric, according to *String* numerical value. Since the name of the DOT files are *etape_0...etape_n* we need *-t* which assigns the field separator, and *-k2* which tells that field 2 should be used. The shellscript generates *animation.gif*. The graph animation can then be viewed in a web browser. Figure 4.1 shows a snapshot of the animation. Enlarged in orange you see a merged state.

Even though the uncustomized animation looked promising, we discovered several disadvantages. Not only that using an external script would be a step back, but also looking at the data bares several disadvantages. For example the reconstructed *ARGStates* have other unique IDs than the original computed *ARGStates*. Another disadvantage would have been possible modifications and customization in already existing *ARG* classes, we wanted to avoid. It was therefore decided not to pursue this approach any further.

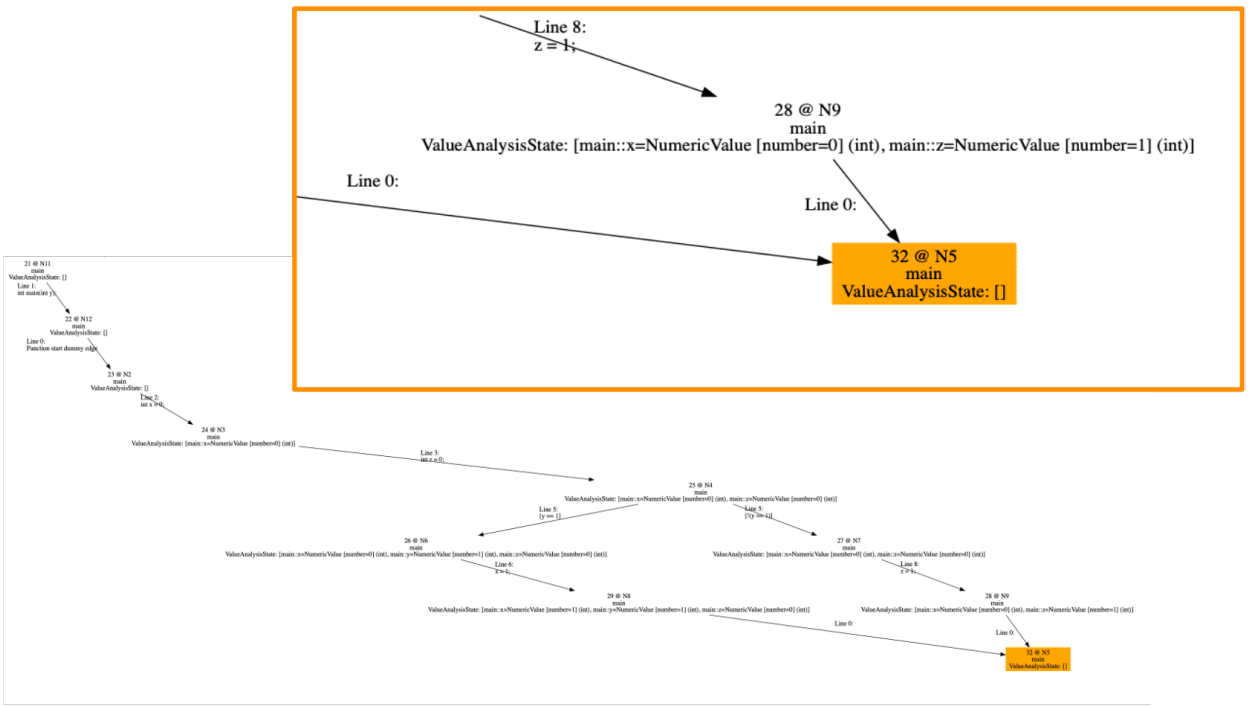


Figure 4.1: snapshot of animation.gif

[3] https://gist.github.com/maelvls/5379127

4.3 ARGStateView

An *ARGState* object can by changed dynamically during the verification run. If it gets destroyed due to e.g. a merge event, the updated *ARGState* has only a diminished information content, namely the ARG ID and the information *destroyed ARG State*. Hence information about predecessors and successors of an *ARGState* are lost. Since it is necessary to store those lost information the java class *ARGStateView* was implemented. An *ARGStateView* object represents an *ARGState* element in that way that it stores the complete set of ARG State information. To initialize the attributes of *ARGStateView* objects, the *ARGStateView* constructor shown in Figure 4.2 takes the parameters *int cCount*, *ARGState cElement*, *Collection<ARGState> cParents*, and *Collection<ARGState> cChildren*. When unwrapping a *CollectorState* we get the potentially already calculated parents (predecessors) and children (successors) of the current *ARGState*, and pass the unmodifiable collections to the *ARGStateView* constructor. These are the lost pieces of information in case an *ARGState* gets destroyed. Additionally an integer *count* is passed. The parameter *count* is required for the chronological order of computation steps. The number is incremented every time a new successor is calculated or a merge is occurred.

```
1  public ARGStateView(
2        int cCount,
3        ARGState cElement,
4        @Nullable Collection<ARGState> cParents,
5        @Nullable Collection<ARGState> cChildren,
6        LogManager clogger) {
7     stateId = idGenerator.getFreshId();
8     element = cElement;
9     count = cCount;
10    wrappedelement = element.getWrappedState();
11    currentID = element.getStateId();
12    if (cChildren != null) {
13      childrenlist = ImmutableList.copyOf(cChildren);
14    }
15    if (cParents != null) {
16      parentslist = ImmutableList.copyOf(cParents);
17    }
18  }
```

Figure 4.2: ARGStateView constructor

Once the method *build()* in the class *CollectorStatistics* is called, the *ARGStateView* getters (Figure 4.3) are used to get the information needed to generate the ARG nodes and ARG edges and to keep the chronological order of the abstract reachability graph. Note that in the preliminary collector version for creating DOT files the *ARGStateView* class was slightly different. For example, a getter was implemented for the wrapped element of the stored *ARGState*, since we needed it for the reconstruction of the *ARGStates*. However the purpose, namely to store information of the wrapped *ARGStates*, was the same.

```
1   public ARGState getARGState() {
2     return element;
3   }
4   public int getStateId() {
5     return currentID;
6   }
7
8   public int getMyStateId() {
9     return stateId;
10  }
11
12   public int getCount() {
13    return count;
14  }
15
16  public Collection<ARGState> getParentslist() {
17    return Collections.unmodifiableCollection(parentslist);
18  }
19  public Collection<ARGState> getChildrenOfToMerge() {
20    return Collections.unmodifiableCollection(childrenlist);
21  }
```

Figure 4.3: ARGStateView getter

4.4 Graph Data

A transformation from a proven as reachable state to a JSON object.

A preferable approach than generating multiple time and space consuming DOT files was to implement methods similar to the already existing ones in the *ReportGenerator*. Briefly sketched the procedure starts with filling two maps one for nodes and one for edges and link them together. They provide all the information needed to create the abstract reachability

graph using Dagre D3. The linked nodes and edges information is passed to the JavaScript in JSON format and the additional HTML file *ComputationSteps.html* is generated.

An advantage of this approach is that it enables to use the *CollectorStates* in the *UnmodifiableReachedSet* directly instead of reconstructing *ARGStates* thereof.

The class *CollectorState.java* is implemented that it saves all information which are necessary for a step by step creation of an abstract reachability graph. By implementing the corresponding *setter* and *getter*, every attribute (e.g. the current *ARGState* ID) of the object *CollectorState* is obtainable. Parents (predecessors) and children (successors) and their attributes are saved in *ARGStateView* and passed to *CollectorState*. In case of a merge the merge partners together with the merged element are saved as *ARGStateViews* in *CollectorState*. While the merged element receives an increased integer for *count*, the merge partners keep their *count* numbers they received from the transfer relation. This ensures the correct chronological sequence. The value for the termination check operator is calculated and assigned as boolean *isStopped* in the *CollectorState*.

The *UnmodifiableReachedSet* is passed to the method *private void build*. The purpose of this method is to assign several variables and to further pass the required information to node and edge creating methods. The assigned *String* variables are *destroyed, toMerge, merged,* and *mergeChild*. More variables are conceivably.

The minimal information for building a graph with Dagre D3 is an index for nodes and source and target indices for edges. Important to mention at this point is the fact that the Dagre D3 library provides the methods to add nodes and edges to the graph object. The function *setNode(arg1, arg2)* receives two arguments. The first one is the node ID, the second is metadata about the node. The node ID has to be unique otherwise only the last node with the same ID will be added to the graph. The function *setEdge(arg1, arg2, arg3)* needs three arguments. The first one is the source ID, the second the target ID. Both are unique IDs of nodes of the graph. The third argument is metadata about the edge. Metadata about nodes and edges are passed as JavaScript objects, containing *name:value* pairs.

The node and edge creating methods in *CollectorStatistics.java* ensure that all necessary arguments are available in JSON format. Two methods for creating nodes are implemented. *createFirstNode* and *createNEWNode*. The first node is treated different, since the implementation does not allow an *ARGStateView* representation of the first *ARGState*. For creating edges also two methods are implemented. While the method *createStandardEdge* is similar to *createArgEdge* in *ReportGenerator* and creates standard ARG edges of the CPAchecker ARG, the second method *createExtraEdge* is implemented for creating edges of computation snapshots missing in the final ARG. The merge operation is such a snapshot. While we want to follow the growing of the abstract reachability graph, meaning the calculation of successors and merge decisions, the final graph only shows the fully computated ARG. Not all

information displayed at the final graph edges remain available during the verification run. Since the information is viewable as edge label after the merge is executed, it is sufficient to label edges only with *"merge edge"* towards elements which will get merged. Edges of nodes that are merged in a future step and point to their successors are marked with *merge child edge*. The *label* of an edge is one of the data, that is passed as metadata in the structure of the JSONobjects. Figure 4.4 shows the complete structure of JSON objects passed to the JavaScript.

```
{"nodes": [
  {"analysisStop":"Boolean x",
  "func":"String x"
  "index":"Number x",
  "intervalStop":"Number y",
  "label":"String y",
  "intervalStart":"Number z",
  "type":"String z"}],
"edges": [
  {"file":"String a",
  "line":"String b",
  "mergetype":"String c",
  "source":"Number a,
  "label":"String d",
  "type":"String e",
  "target":"Number b"}]
}
```

Figure 4.4: Structure of the Collector JSON object

The collector JSON object contains two keys (*nodes* and *edges*), each containing arrays of objects. Nodes and edges pass their information via those arrays of objects.

An information about the termination check of the current state, respectively node, is passed in form of a Boolean *analysisStop*.

The function in which the node is present is passed by a String (e.g. *"main"*) as value for the key *func*.

The key *index* represent the unique ID of the current state and is passed as *int*.

For the step by step evolution of the ARG we want to use information of "how long is a state alive?". The fact that merge partners are alive as long as they have not been merged leads to the solution to pass the *count* number of the merged element as value for *intervalStop* of the merge partners. If the element is alive infinite the value is *" "*. The key *intervalStart* is the *count* number of the current element. How exactly this information is used for building the graph is described in section 4.5.

Label and *type* provide the same information as the original ARG.

The metadata for *edges* also provide the same information for *file, line, label,* and *type* as the original ARG.

Source and *target* are the indices of the nodes they connect. They are passed as *int*.

The key *mergetype* is integrated in the JSON object for marking the edges towards states which will get merged in the future of the evolving ARG. The values are either *none*, *toMerge* for merge partners, *mergeChild* for successors of a mergepartner state, and *merged* for the node representation of a merged state.

After the nodes and edges lists are filled and linked, the method *makeHTMLfile* is called. It generates the file *ComputationSteps.html* in the output folder of CPAchecker. HTML, JavaScript and CSS are combined and the JSON data is written using the nodes and edges information out of the *LinkedHashMap*. The implementation of those three parts is described in the following section.

4.5 Web Frontend

To view and interact with data provided by the newly implemented collector CPA, a new web frontend using HTML, CSS and JavaScript was developed. The separate implementation of *collector.html, collector.css*, and *collector.js* was indicated due to the Separation of Concerns principle in software development. The HTML code provides the framework of how the site will look like and includes action-causing strings as *meta* tags. The presentation style of the elements on the site is controlled by the CSS code and finally the event-based JavaScript code makes the site interactive by manipulating the data in response to events (e.g. clicking buttons or operating the slider). Figure 4.5 shows a screenshot of the web frontend. The functionality will be explained in detail later in this section.

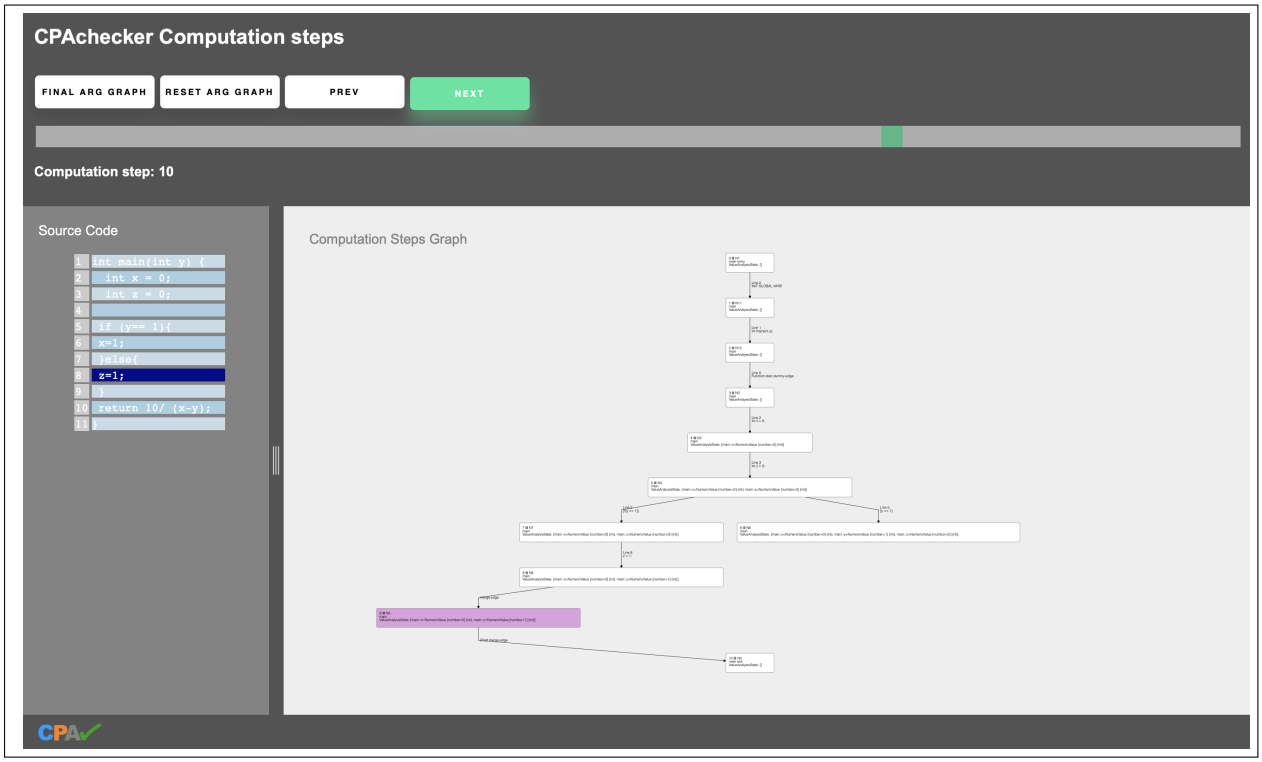


Figure 4.5: Screenshot of the web frontend using Safari

As soon as the *makeHTMLFile* method in *CollectorStatitics.java* is called, the HTML template is read line by line. The included *meta* tags are <!−−COLLECTOR_CSS−−>, <!−−SOURCE_CONTENT−−>, and <!−−COLLECTOR_JS−−>. The action which is caused by the first *meta* tag is the inclusion of the CSS code. Reading the *meta* tag <!−−SOURCE_CONTENT−−> results in inserting the source code of the analyzed C-program as a table-like representation. The method *insertSource* was implemented in *CollectorStatistics* similar to the *insertSource* method in the *ReportGenerator*. The *meta* tag <!−−COLLECTOR_JS−−> leads to the integration of the JavaScript code. Reading COLLECTOR_ JSON_ INPUT in the integrated JavaScript code effects in passing the graph data in JSON format. The graph data is the only dynamic data used in the creation of this additional output content (namely *ComputationSteps.html*) of the CPAchecker. It is prefixed with the declaration of the JavaScript variable *var data*.

Beside the function *dagreGraphBuild* the JavaScript *collector.js* has implemented several functions for evolving and manipulating the graph interactively. The first step in creating the graph using Dagre D3 is the assigning of an empty graph object. The second step is to add the nodes and edges to the empty graph (Figure 4.6) .

```
data["nodes"].forEach(function (v) {
        g.setNode(v["index"],
        {label: v["label"],
         class: v["type"],
         id : "node" + v["index"]});
        });

data["edges"].forEach(function (v) {
        g.setEdge(v["source"], v["target"],
        {label: v["label"] ,
        class: "source" + v["source"]+ " "+ "target" + v["target"],
        id:"source"+ v["source"] + "target" + v["target"],
        labelId: "source"+ v["source"]+"target"+ v["target"]})
        });
```

Figure 4.6: Setting nodes and edges

The structure of each node and edge is chosen in that way, that it is possible to use the CSS class and the ID of nodes and edges to select and thereby control the manipulation of the individual SVG elements. Setting up a SVG group is required for translating the final graph. This is done by calling the Dagre D3 renderer. At this point some layout properties (e.g. rounding of node corners) are configured and the zoom support is implemented (Figure 4.7)

```
var zoom = d3.zoom().on('zoom', function () {
        transform = d3.event.transform;
        svgGroup.attr("transform",
        "translate("+ transform.x + ", " + transform.y +")
        scale(" + transform.k + ")");
    });
    svg.call(zoom);
```

Figure 4.7: Zoom support

```
1 var t = d3.zoomIdentity.translateBy(tx, ty).scale(k);
```

Figure 4.8: Pan support, move your target into a position

The zoom support is complemented by the pan support. D3 provides a transform producing function, where you can determine the new position(*tx,ty*) and scale (*k*), see Figure 4.8. A step by step manual for the implementation of zoom and pan you will find on the freeCodeCamp site [4].

Now since the graph is built, the manipulation could start. Some of the nodes and edges have beside their default Dagre D3 CSS classes (*edgepath, edgelabel* and *node*) already assigned CSS classes to distinguish them from the beginning on. To mark prospective merge partners and merged elements, nodes obtain the value of its type as CSS class (e.g. the value *toMerge*). Beside the implemented CSS color *plum* for merge partners and *turquoise* for the merged element, a dashed node rim is implemented for the termination check boolean *analysisStop*.

To select specific edges, the predefined CSS class of an edge is a String composed of *source* plus the corresponding index number. A second additional class for edges is *target* plus its index number (see Figure 4.6). With the CSS class selector you can then select elements with specific class attributes and subsequently manipulate them.

Whereas multiple elements in a document can have the same class value, the CSS ID must be unique. The unique CSS ID of a node is composed of string *node* and its unique ID number. For edges it is a concatenated string *source* + x + *target* + y, where x and$y$ are the corresponding index numbers of the source and target nodes (see Figure 4.6).

For the step by step evolution of the ARG in the right time sequence the JavaScript has implemented a function to sort the nodes in chronological order using the passed *count* respectively *intervalStart* value (see Figure 4.9).

```
1 myVar["nodes"].sort(function (a, b) { return parseInt(a.intervalStart)-
    parseInt(b.intervalStart) });
```

Figure 4.9: Sorting the nodes in chronological order

[4] https://www.freecodecamp.org/news/get-ready-to-zoom-and-pan-like-a-pro-after-reading-this-in-depth-tutorial-5d963b0a153e/

When the user opens the *ComputationSteps.html* document, the function *start()* calls the *shownode* function with the first element of the graph as parameter. The first node receives the CSS class *contentshow*. All nodes and edges have the CSS attribute *visibility : hidden*. Selecting by the class *contentshow* sets the visibility value to *visibility : visible*. Note that at this point the whole graph is built with Dagre D3, but the user only sees the first node due to the assignment of visibility of each graph element via its CSS class, meaning the first node is set to *visible*, whereas all remaining nodes still have the attribute set to *hidden* . Figure 4.10 shows an example of how the starting page looks like. On the left side the source code of the analyzed C-program is displayed. On the right side the user will see the evolving ARG, when clicking the NEXT button or operating the slider.

Figure 4.10: Start page

The user can follow the evolution of the ARG in the *Computation Steps Graph* box on the right with the implemented possibility to zoom and pan the graph (Figure 4.11). On the left the source code line corresponding to the edge pointing to the current node is highlighted in darkblue. D3 selection and class attribute flipping is implemented similar to the function *markSourceLine* in *report.js*. Right-handed in green (Figure 4.11) you see the implemented mouse over tool tip box, which provide information about the ARG ID and label of the node.

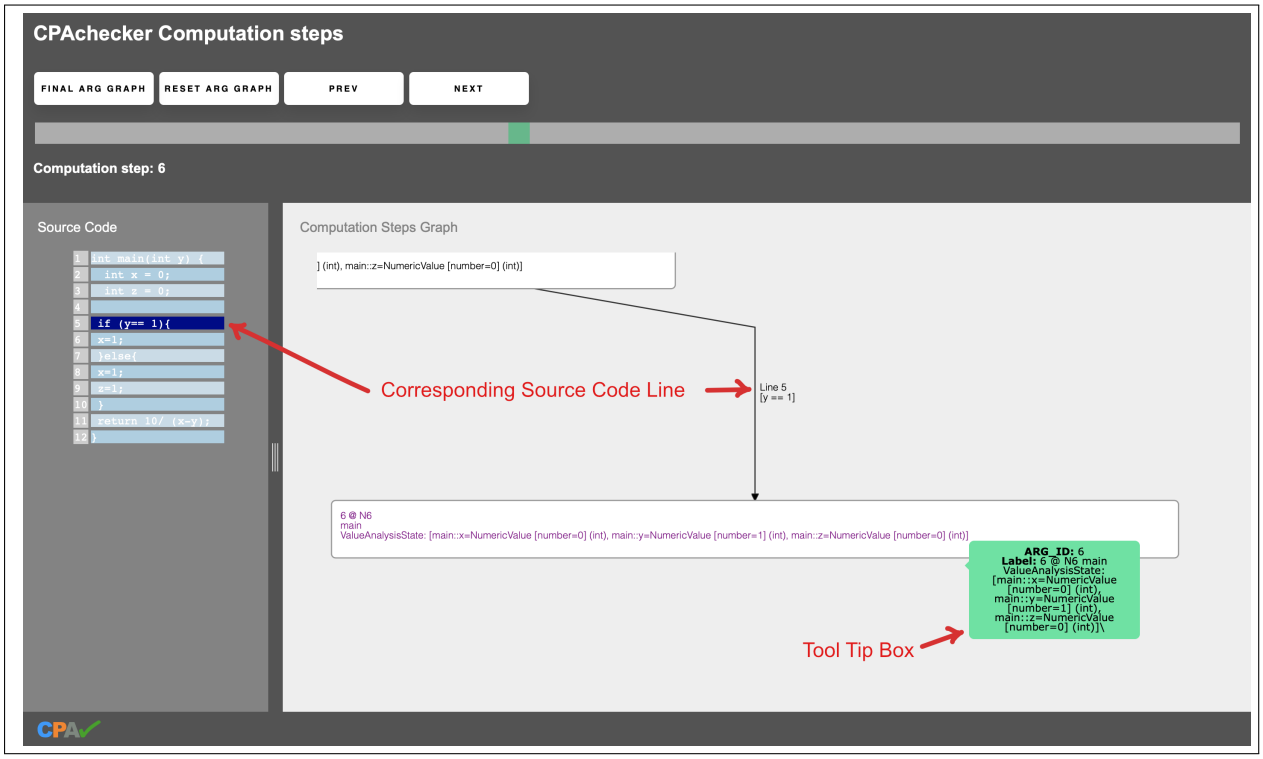


Figure 4.11: Zoom and highlighted source code line

In the following, I will describe further details of each HTML element and how the implemented *collector.js* functions control the behavior by using the jQuery library (section 3.2.3). The control of the behavior is fundamental to observe the structure of the graph step by step.

Flexbox

Since vertical centering of elements in a container is a common difficulty in HTML the Flexbox module was used. Flexbox is a CSS based container layout module or in other words a container manipulation tool. The page layout with nested Flexbox containers was implemented by following one of the examples provided on this web log site[5]. It allows a side by side layout of graph and source code of the analyzed C-program. A small jQuery plug-in[6] was used for the implementation of a splitter pane. The user has thus the possibility to resize the *Source Code* box (on the left) and the *Computation Steps Graph* box (on the right) depending on his current needs.

[5] `https://weblog.west-wind.com/posts/2017/Nov/11/Flexing-your-HTML-Layout-Muscles-with-Flexbox`
[6] `https://github.com/RickStrahl/jquery-resizable`

PREV and NEXT

Mouse clicking events on the *NEXT* button result in the function call *nextStep()*. A step counter will be increased for each click event. In case the current step number is smaller then the maximum step number the function *shownode* is called with the current step number. The function iterates over each node in chronological order until the current node is reached. Depending on their CSS classes and their unique IDs, the jQuery selector is used to manipulate the behavior and appearance of each node and edge. Figure 4.12 and 4.13 show two snapshots of an evolving ARG. The snapshots are suitable to demonstrate the different behavior due to CSS classes and IDs. The plum colored nodes in Figure 4.12 are elements with three relevant CSS classes. The predefined class *toMerge* which is responsible for the color, the class *contentshow* is assigned as visible, and the class *Stop14*, which has no style property. The class *Stop14* is a concatenation of the String *Stop* and the value of the variable *intervalStop*, which is in that case 14. It classifies the node with the information that this node is alive (visible) until the value *intervalStart* of a future current node equals its *intervalStop*. In Figure 4.13 the value of *intervalStart* of the current node is equal to *intervalStop* of the two plum colored nodes in Figure 4.12. Meaning in the first snapshot you see the two merge partners and in the next computation step (Figure 4.13) the merge is calculated. The merge partners are selected using the fitting jQuery class selector and their CSS class *contentshow* is removed. Subsequently the merge partners disappear due to the value *hidden* for those nodes. The merged element has at this point the relevant CSS class *mergedColored* to identify the node as merged element colored in cyan. The successor, which was already calculated for one of the merge partner is transferred to the merged element as well as the predecessors (compare Figure 4.12 and 4.13). Edges to nodes, representing states which will get merged are labeled with *merge edge*, whereas the edges to finally merged element are labeled with the original ARG label. Edges starting from a merge partner to an already calculated successor are labeled with *child merge edge*. After the child is passed to the merged element the edge gets the original ARG label.

Clicking the button *PREV* causes a decrease of the step counting variable and in term calls the function *shownode*. The same function as triggered by clicking the *NEXT* button. Since the function is called with the counting value (*step-1*) the iteration ends one step prior. The implementation allows the user to go back and forth in the computation of the evolving ARG by using the *PREV* and *NEXT* buttons.

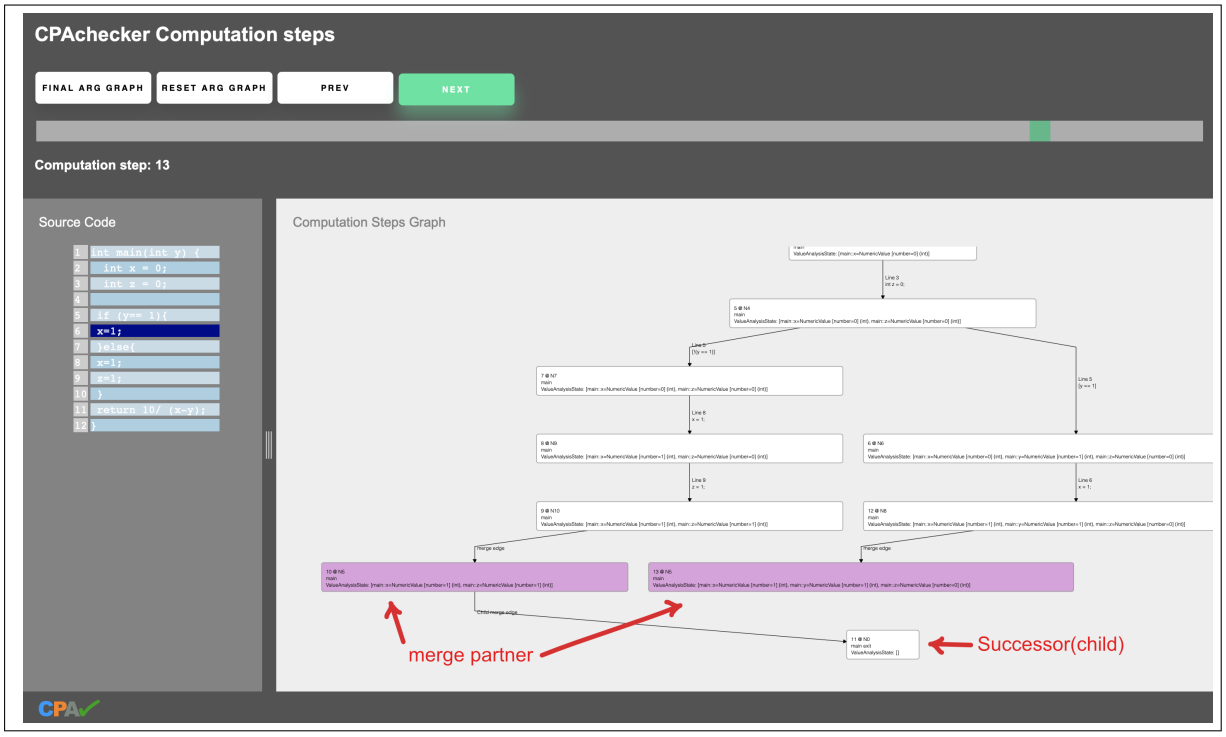


Figure 4.12: Snapshot of evolving ARG before a merge

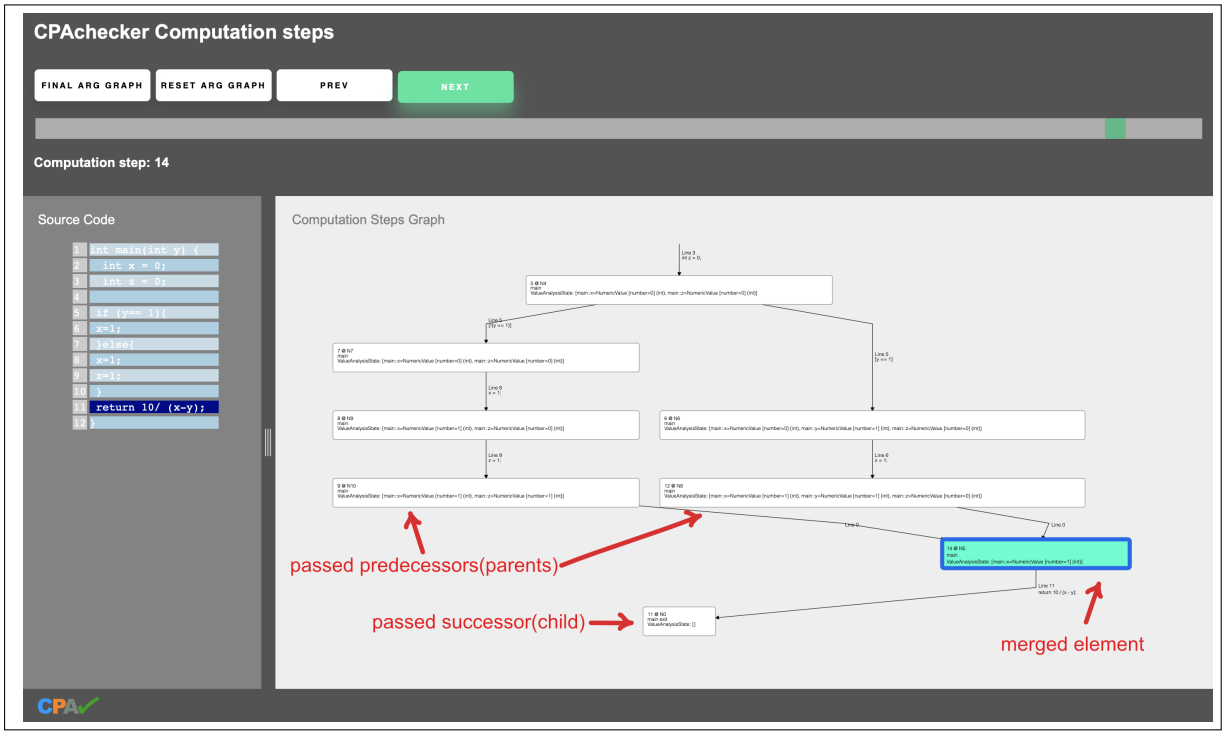


Figure 4.13: Snapshot of evolving ARG with merged element

Slider

The back and forth functionality is also implemented for the slider. The function *slide()* requires information about the current computation step, which is constantly updated. It gets the information about the current step by the common *getEementById()* method for HTML DOM selection and manipulation of elements. Subsequently the function *shownode* is called with the value of the current step.

RESET

The reset button is a simple implementation divided in two commands. First the step counting value is set to *step = 0* (slider and computation steps display) and second the function call *start()*.

FINAL ARG

The user has the possibility to view the results of the verification analysis in form of an abstract reachability graph, when choosing *STANDARD ARG* in the drop-down list of the *FINAL ARG* button (Figure 4.14). This function is implemented that it calls the *shownode* function with the maximum computation step of the verification run. Choosing *HIGHLIGHT MERGE* calls another JavaScript function, where all nodes and edges are selected and the CSS class *contentshow* is added. Furthermore the merged elements are selected by their predefined class *merged* and by adding the class *mergedColored* the merged elements are also highlighted in the final graph (Figure 4.15).

Figure 4.14: Standard final ARG

Figure 4.15: Final ARG with highlighted merge partners (plum) and merged elements (cyan)

5 Evaluation

In order to determine whether the implementation of the new web frontend has the benefits we had hoped for, a user survey was designed. Gathering feedback from the people that will be profiting from the software is done by sending a survey to those people. The evaluation survey (A.1) itself was created using Google forms and distributed via a link. This section describes the evaluation concept (5.1) and its results (5.2).

5.1 Evaluation concept

The aim of the survey was for one thing to evaluate the use of the new web frontend as a tool for students to compare and correct their exercise results of manual creation of an abstract reachability graph using the CPA algorithm, and secondly as a tool for the general user to improve his understanding of the analysis results.

The evaluation consists of four parts, a general questionnaire, two tasks followed by specific questions, and a part for general ratings. The general questionnaire was conceived to find out how well the participants know the CPAchecker and how often they use the different features of the CPAchecker report. The first task served to familiarize the participants with the new web frontend. The participants were asked to perform the analysis with the newly implemented Collector CPA and the already shipped example C-program of the CPAchecker. This example was used because it shows several merge events. After the participants had completed the analysis, the task was to test all features. Based on the experiences of the participants, the questions to be answered should give me an idea of the operability. In addition, ideas for improvements should also be collected. The second task was designed to find out if the new web frontend is suitable to make the calculation of the merge operator easier to understand. To keep the effort for the participants as low as possible it was necessary to choose a simple C-program. The participant should be able to easily follow the requirements of typical student exercises. A typical exercise for student is: "Perform a CPA analysis on the given program (represented by the CFA) and represent the resulting set of reachable states as an abstract reachability graph (ARG)." A slightly modified program from the Handbook on Model Checking [4] was suitable for this task (Figure 5.1). The CFA of this C-program has eight program locations ($L = \{2, 3, 5, 6, 8, 9, 11, 12\}, l_0 = 2$) and three program variables

($X = \{x, y, z\}$). At location 6 there is only one concrete state reachable from the initial region. The variable assignment at this location is $c(pc) = 6, c(x) = 0, c(y) = 1, c(z) = 0$. The set of concrete states at program location 11 can be represented by the predicate $pc = 11 \wedge ((x = 1 \wedge y = 1 \wedge z = 0) \vee (x = 1 \wedge y \neq 1 \wedge z = 1))$. The variable assignment of the two concrete states are $c(pc) = 11, c(x) = 1, c(y) = 1, c(z) = 0$ and $c(pc) = 11, c(x) = 1, c(z) = 1$. The merge operation will merge those two concrete states to a more abstract state with variable assignment $c(pc) = 11, c(x) = 1$. To solve this typical student exercise the students need to calculate the merge manually. Before they are able to draw the final ARG, they had to calculate the states which will get merged. Using this rather simple example makes it easier or faster to solve the exercise if the participant wants to. However the task was designed that the participants do not need to calculate the ARG manually, but being able to understand how the calculations are done. The questions that should be answered afterwards are planned to help to draw conclusions, whether the new web frontend can serve as a tool for a better understanding of the calculations of the CPAchecker. Finally, the participants are asked to rate the usability, the web design, and their user experience.

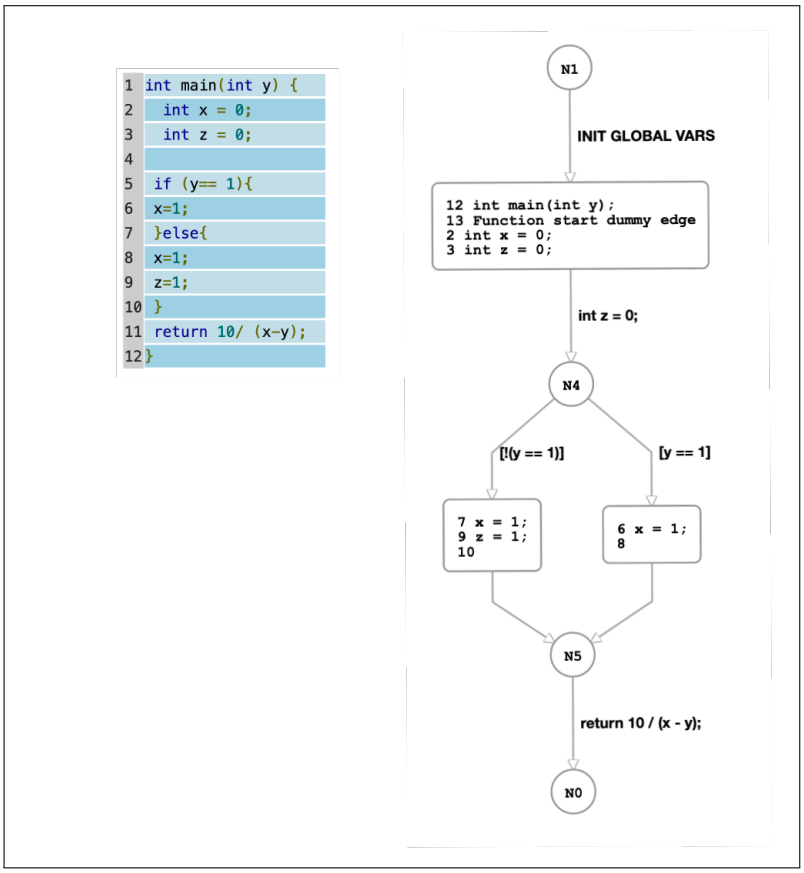


Figure 5.1: Source code of the C-program and corresponding CFA

5.2 Evaluation results

Participation in the survey required a minimal knowledge of CPAchecker. For this reason, the pool of participants and the response rate to the survey was rather low. A quantitative statement can therefore not really be made. Nevertheless, the results are presented and discussed here. A summary of all answers as generated by Google can be found in the appendix A.2. Based on the statements that the participants are familiar with the CPAchecker and use the report weekly to monthly, it can be assumed that the answers are of high quality.

The participants were asked how often they use the different features of the report. The answer scale ranges from 1 to 5, where 1 means *every time* and 5 *never*. It turns out that 50% of the participants uses the ARG-tab every time. The remaining 50% rated it at 2. 50% of the user also rated the CFA-tab usage at 2, the rest of the users rated equally at 1 and 3. The remaining features are used less. The frequency of use in descending order starts with the source-tab, followed by log-tab and configurations-tab and as final the statistics-tab. The usage of interactive features of the report like clicking on edges of the CFA to switch to the source code line or clicking on elements of the ARG is median distributed. 75% of the participants regularly used the outdated ARG.dot files. If they still use the ARG.dot files can unfortunately not be derived from the answers.

I tested the new web frontend using several common browsers. Namely Mozilla Firefox, Safari, Internet Explorer, and Google Chrome. The participants of the survey did not use any other browser. As expected no display problems were discovered. The criticized contrast of the source code view is independent of the browser choice. By changing the font color from white to dark blue on light blue background the point of criticism was immediately fixed. In summary task 1 "Testing the features" resulted in more or less what was expected. All Buttons, the slider, the hyperlink, the displayed graph, the tooltip box and the pan and zoom function worked to the fullest satisfaction. One outlier of the slider performance is due to mix-up of splitter and slider, as I assume by looking on each survey response separately. One of my own criticisms where I was undecided was the width of Computation Steps Graph box. The splitter has the function to resize the boxes. As one participant was also not entirely satisfied I decided to change the min-width from 150px to 25px in the CSS file. Another not quite satisfying fact was that the source code highlighting is not as obvious as I thought. Only 50% of the participants rate it as *perfectly obvious* that the highlighted source code line matches the edge label. The content of the tooltip box was pretty much clear. The only suggestion for additional displayed data was to add "The step were the state is created or merged". The improved tooltip box now shows beside the ARG ID and the label of the node, information about the step when the node is created, the type of the state and the step when it will be merged. The implemented improvements (font color of source code and additional information in tooltip box) are shown in Figure 5.2.

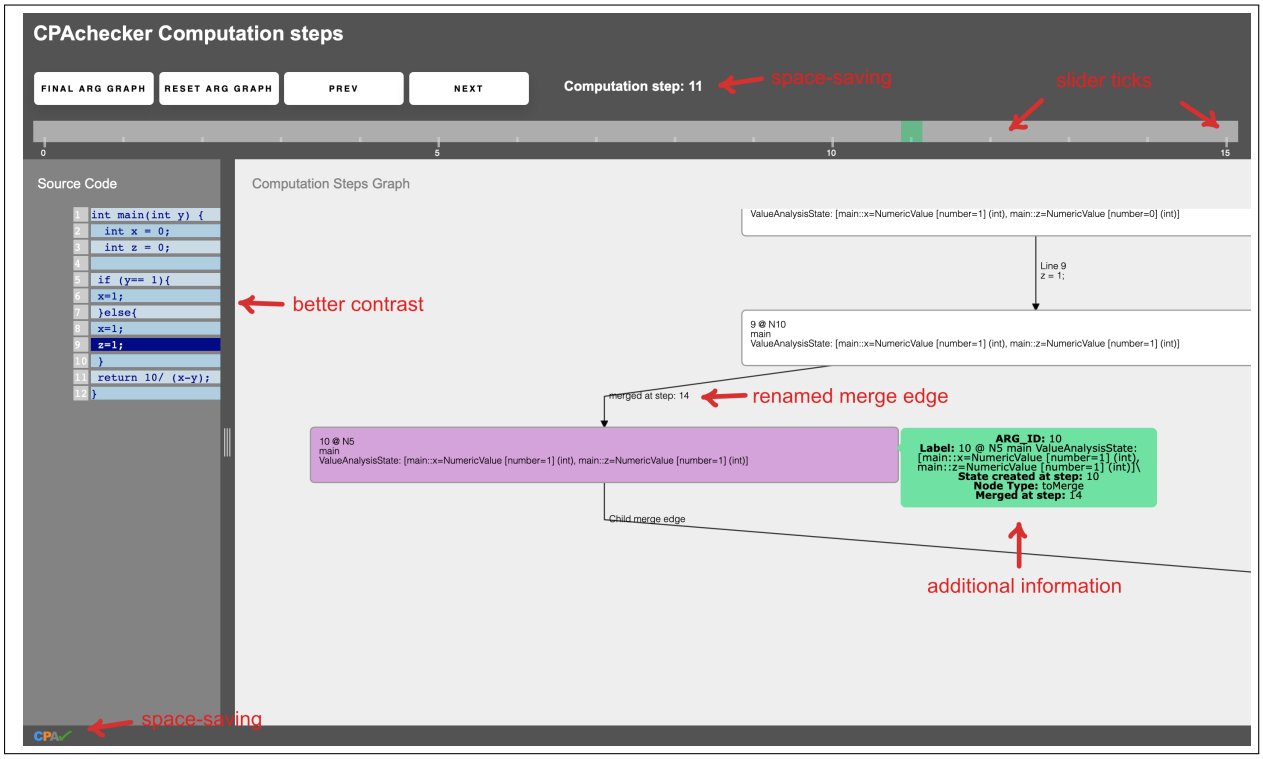


Figure 5.2: Implemented improvements

The results of task 2 of the survey should answer the question if the new web frontend is suitable to make the calculation of the merge operator of the CPA easier to understand. The implementation makes it easy to find or go to specific steps and nodes. All participants said it was clear how to get to a specific step. 50% rated it as supereasy (mark 1) to find a specific node and 50% rated it with mark 2.

The node color of merge partner states in the graph help to understand the merge operation said 75% of the participants (mark 1). The remaining 25% rated it with mark 2. Even though merge partner nodes and merged element are on the same vertical graph level it was not absolutely clear that those states are get merged. 75% rated it with mark 2 and 25% with mark 3. The second task was structured in such a way that the participants had to put themselves in the position of a student in order to answer the question which features of the new web frontend help to understand the calculation steps of the CPAchecker. All participants are of the opinion that the final ARG *Highlight Merge* will help a student to correct or compare his manually calculated results. 50% think that also the final standard ARG is of help. On average they rated the question "Does the final ARG *Highlight Merge* made the computation steps of the CPA Algorithm clearer?" with mark 2, where 1 was *Yes, absolutely* and 5 *Not at all*. All participants of the survey are convinced, that the possibility to go back and forth by using either the PREV/NEXT buttons or the slider help

to understand the computation steps of the CPA algorithm. But only 50% are convinced that highlighting the source code line leads to a better understanding of the computation steps. The final evaluation questions whether the new web frontend is useful for students and general users were rated by 50% of the participants with grade 1 and 50% with grade 2.

The last part of the survey consisted of three general rating questions about usability, web design and user experience. The scale for the usability is 1 for *Completely unclear where to find and click* and 5 for *Everything was self-explaining*. 75% of the participants rated it with mark 4 and 25% with mark 5. The scale for the web design was 1 for *not attractive* and 5 for *attractive*. 100% gave a solid 4. The scale for the user experience ranged from *It was a pain working with it* (1) to *I enjoyed working with it* (5). 50% enjoyed working with the new web frontend. The other half rated it with mark 4.

Finally the participants had space for suggestions, wishes, comments, and remarks. My thanks to all participants for the large number of answers listed here:

- If you loose the graph while zooming and paning a "find" or "focus" Button would be nice to reset the view
- Missing features from other report (clicking on edges, clicking on source code).
- Highlight the states on the waitlist! It was confusing in the last example in steps 7-11 because I did wonder whether 6@N6 would still be explored, i.e., whether it is still on the waitlist or not. The slider does not go all the way to the left. The top bar takes a lot of screen space but cannot be minimized. The Computation Steps Graph Windows does not move as I would expect it when moving the slider. I would expect the Graph to not move at all when I move the slider, but the slider simple moved the whole canvas with it. So I need to readjust/recenter the graph after using the slider. I also do not see whether a state was stopped (coverag) and if so by which state. I really like the fact that all states at the same location/ that are merged are on the same graph rank (vertical position). Makes it easier to understand what is happening. Maybe that is just the case in the examples. This could make the graph harder to read if the program is more complicated. There is also a projected view for analyses that use ABE. Most of the ARG nodes are not used for merge there. Supporting this would make ARGs for analyses that use ABE better understandable.
- "The contrast of the source-code view is a little bit low, white on light blue is too hard to read. Syntax highlighting like in the standard report would also be nice.

  In general: Why have a separate file? Can't the two HTML files be merged, such that the interactive ARG is a tab on the existing report? Or even replaces the existing ARG tab? This would also solve the problem that the current ComputationSteps.html

does not provide any further details about the verification, e.g., which program file was analyzed, which version of CPAchecker, which configuration, etc.

In the first file I opened (for example.c) I had tooltip contents that overflowed the tooltip box.

The term "Highlight merge" is not clear to me.

In the "Highlight merge" view it would probably make sense to actually connect the merged states with its resulting state, e.g., by adding dashed edges or so.

I am not sure whether it is helpful that all states that will be merged at some point in the future are immediately highlighted when they are added. This can be confusing when going step-wise and a new highlighted node appears without explanation why it is highlighted. Maybe highlight them only in the last step before the merge?"

- The CPAchecker-Logo causes 1cm of space usage at the bottom. This is an inacceptable waste of space. Can you insert ticks for the computation steps slider? Can you rename the "merge edge" or insert additional edges "merged into"? This would make thinks clearer.

As already mentioned, both the minimum width of the graph box and the contrast of the source code line were immediately improved. Another possibility to improve the contrast of the source code line is to initialize Google pretiffy code as it is done in the report (Figure 5.3). This would also satisfy the wish of highlighting the source code syntax. I classify that as matter of preference and decided to go for less color instead of syntax highlighting.

```
1 // Initialize Google pretiffy code
2 $(document).ready(function () {
3   PR.prettyPrint();
4   });
```

Figure 5.3: Syntax highlighting

A previously unnoticed problem, that the tooltip content overflowed the tooltip box is solved by a change of width size. As already mentioned the tooltip content itself has now additional information, among other things about the step where a state will get merged. Since one suggestion was to rename *merge edge* I decided to add this information (merged at step x) also to the merge partner edges (see Figure 5.2).

Although I disagree with the comment that the size of the CPAchecker logo is an unacceptable waste of space, since a non-participant in the survey missed the hyperlink, I have minimized it by 40% as a compromise.

Another criticism about space was that the top bar can not be minimized. I am aware of this, but see it as necessary that the buttons, the slider and the Computation Steps display remains permanently visible. But to save at least a bit of space I put the Computation Step display on top, next to the buttons (see Figure 5.2). Space and size is also a problem for big graphs of more complex programs. The visualization does not work well for very large programs, but the goal is teaching and understanding in small programs, so it was decided to focus on that.

For faster access to certain calculation steps, ticks were attached to the slider as suggested by one survey participant. They have been implemented so that they adapt dynamically to the number of calculation steps (see Figure 5.2).

One suggestion was already implemented, but the program examples in the survey did not focus on that. The user will see stopped states as nodes with dashed borders in case the boolean *analysisStop* is true (see Figure 5.4).

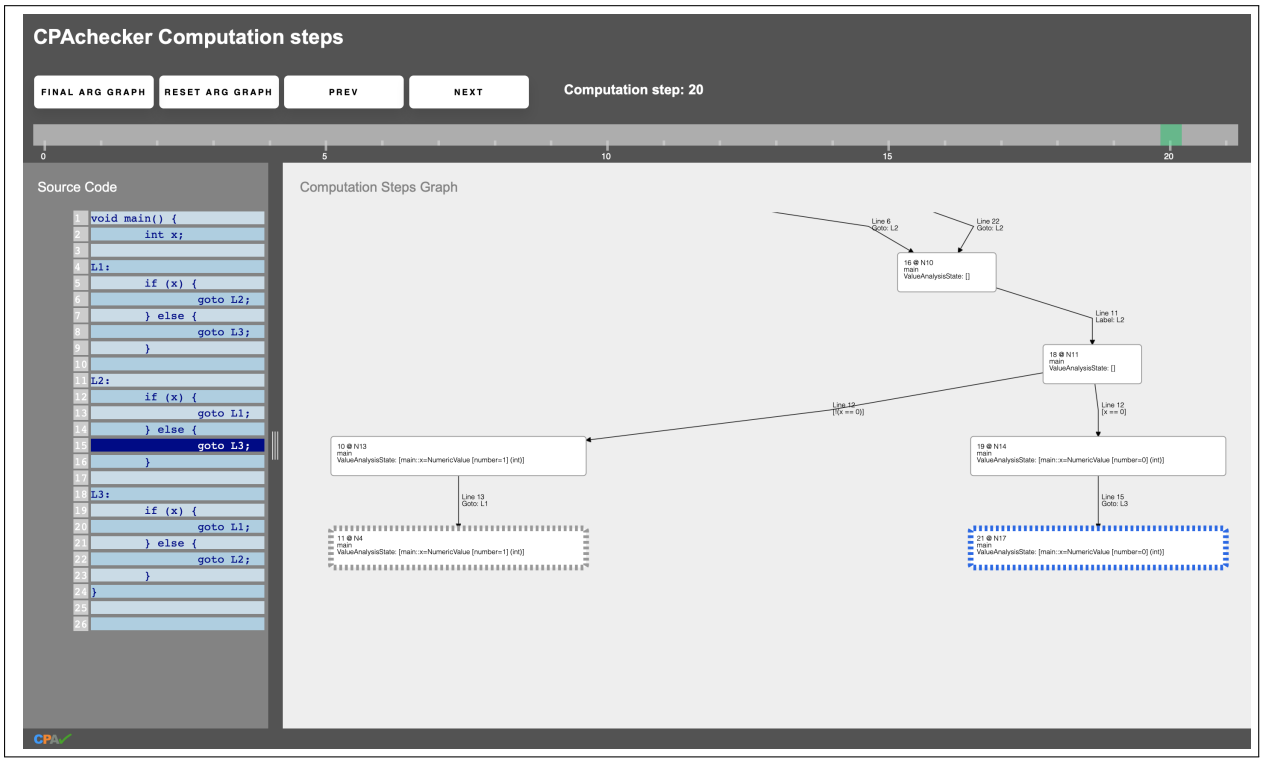


Figure 5.4: Dashed node rim of stopped states

Since the remaining suggestions (additional edges, integration of the new web fontend in the report, find button, supporting ABE analyses) provide great ideas for future work, they will be discussed in the next chapter (6).

6 Future Work

During implementation of the new web frontend and discussions with my advisor we came up with multiple desirable ideas for future work. Some of them were also listed as wishes and suggestions from the survey participants.

Integration in the report

We started with the idea to implement an additional output for the purpose of visualizing the computation steps of CPAchecker. It turned out that more and more features of the already existing report would also be nice in the *ComputationSteps.html*. The best solution could be to integrate it into the report file as extra feature or by replacing the actual ARG tab. It will be part of future work to decide whether an integration makes sense or whether additional features like configuration and statistics should be integrated into the separate *ComputationSteps.html*. Depending on what you decide, either the help button in the report must be extended or a help button must be inserted. One of our considerations was at what point should the merge partners be highlighted in color. A mix of the improvements already made plus a help function should get these ambiguities out of the way. A help function can also indicate that the current edge label matches the highlighted source code line. Irrespective of this, future work should focus to improve the new web frontend by highlighting all calculation steps. To name a few: highlight states on waitlist, highlight states before and after precision adjustment, highlight source code (pseudocode) of the CPA algorithm.

Additional ARG edges

A problem I could not solve to my full satisfaction was the question "Are extra edges from merge partners to the merged element helpful or confusing?". In case one think it is helpful would mean an additional calculation step in the CPA algorithm. The CPAchecker does not compute edges of merge partners to their computed merged element. The current implementation creates as a first step the complete final ARG with dagre D3, where beside the first node all other nodes are invisible. It is therefore not a real live recording of the ARG

creation but rather a simulation by changing the visibility of the nodes. Finding a better solution for that could therefore be part of a future work.

Quantitative Evaluation

Even though the website is well received by the survey participants, there are many things that need to or could be improved. The web design can be easily adapted at any time, but it is a challenge to optimally arrange all content in a limited space. Future work will probably achieve a better arrangement. It is also worth discussing whether a find/focus button to reset the view in case you lost the graph by panning and zooming is mandatory or not needed. Another point of discussion could be if users prefer that the graph is not moving when the splitter is used or if the majority of the users prefer that the whole canvas with the graph fixed is moving like I do prefer. A larger number of survey participants could have answered questions of this kind. To get a more quantitative result in this respect, testing the new web frontend with students under real conditions where they have to solve the typical exercise of manually calculating an ARG as before and then check their result with the new web frontend could be part of the future work.

ABE Support

ABE (Adjustable-Block Encoding) was introduced in CPAchecker to obtain a better precision and performance of the verification analysis [12]. Supporting this with the the new CollectorCPA and web frontend could be a nice future project.

7 Conclusion

This thesis provides a successful design and implementation of a new web frontend for visualization of computation steps and their results in CPAchecker. The work was divided in two implementation parts. First implementation of a new CPA and second the implementation of the new web frontend. The storage of states which get lost during the verification analysis but being important for a comprehensive understanding of the verification results was achieved by implementing the new CPA *CollectorCPA*. This new CPA acts as a wrapper of the ARG CPA. Calculations like merge and termination check are all delegated to the operators of the ARG CPA, but stored in the Collector CPA. Since the visualization of complex logical calculations makes a large contribution to the understanding of the same, emphasis was also placed on the design, choice of visualization elements like colors and graphics, and features for interactivity. The survey showed that we have made a suitable selection of those visualization elements. The automated generation of an additional output *Computation-Steps.html* is achieved in the same way the report is generated. The fact that the HTML page is cross-browser compatible was also confirmed by the survey participants. The data produced by the CPAchecker or more precisely by the Collector CPA for creation of the interactive ARG is provided dynamically and structured in JSON format to the HTML file. The already established use of Dagre D3 was a good choice to create the final ARG. Starting from this an interactive platform was implemented in form of a routine in JavaScript. The new web frontend was completed with HTML for the framework and CSS for the style. The subsequent evaluation showed that all implemented functions work to our full satisfaction. Furthermore the participants of the survey were convinced that the new web frontend is a great help not only for students but also for general users. The most important achievement in respect for an enhanced understanding of computation steps of the CPAchecker is the possibility to go interactively forth and back in the step-by-step creation of the ARG, since 100% of the survey participants rated it as absolutely helpful. In summary, it can be said that the design and implementation of this new web frontend achieves a better comprehension of the calculation steps and results of CPAchecker. The new web frontend provides a good basis for further improvements, supports and integration in the report of the CPAchecker.

Bibliography

[1] Gleiss, Bernhard ; Kovács, Laura ; Schnedlitz, Lena: Interactive Visualization of Saturation Attempts in Vampire. In: Ahrendt, Wolfgang (Hrsg.) ; Tarifa, Silvia Lizeth T. (Hrsg.): *Integrated Formal Methods - 15th International Conference, IFM 2019, Bergen, Norway, December 2-6, 2019, Proceedings* Bd. 11918, Springer, 2019 (Lecture Notes in Computer Science), 504–513

[2] Zhou, Chijin ; Wang, Mingzhe ; Liang, Jie ; Liu, Zhe ; Sun, Chengnian ; Jiang, Yu: VisFuzz: Understanding and Intervening Fuzzing with Interactive Visualization. In: *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*, IEEE, 2019, 1078–1081

[3] Beyer, Dirk ; Keremoglu, M. E.: CPAchecker: A Tool for Configurable Software Verification. In: Gopalakrishnan, Ganesh (Hrsg.) ; Qadeer, Shaz (Hrsg.): *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings* Bd. 6806, Springer, 2011 (Lecture Notes in Computer Science), 184–190

[4] Beyer, Dirk ; Gulwani, Sumit ; Schmidt, David A.: Combining Model Checking and Data-Flow Analysis. Version: 2018. `http://dx.doi.org/10.1007/978-3-319-10575-8_16`. In: Clarke, Edmund M. (Hrsg.) ; Henzinger, Thomas A. (Hrsg.) ; Veith, Helmut (Hrsg.) ; Bloem, Roderick (Hrsg.): *Handbook of Model Checking*. Springer, 2018. – DOI 10.1007/978–3–319–10575–8_16, 493–540

[5] Beyer, Dirk ; Henzinger, Thomas A. ; Théoduloz, Grégory: Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis. In: Damm, Werner (Hrsg.) ; Hermanns, Holger (Hrsg.): *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings* Bd. 4590, Springer, 2007 (Lecture Notes in Computer Science), 504–518

[6] Zakharov, Ilja S. ; Mandrykin, Mikhail U. ; Mutilin, Vadim S. ; Novikov, Evgeny ; Petrenko, Alexander K. ; Khoroshilov, Alexey V.: Configurable toolset for static verification of operating systems kernel modules. In: *Programming and Computer Software* 41 (2015), Nr. 1, 49–64. `http://dx.doi.org/10.1134/S0361768815010065`. – DOI 10.1134/S0361768815010065

[7] Takanen, Ari ; Demott, Jared D. ; Miller, Charles: *Fuzzing for Software Security Testing and Quality Assurance*. 2nd. USA : Artech House, Inc., 2018. – ISBN 1608078507

[8] Rockai, Petr ; Barnat, Jiri: A Simulator for LLVM Bitcode. In: *CoRR* abs/1704.05551 (2017). `http://arxiv.org/abs/1704.05551`

[9] Wendler, Philipp: *Towards Practical Predicate Analysis*. PhD Thesis, University of Passau, Software Systems Lab. `https://www.sosy-lab.org/research/phd/wendler/`. Version: 2017

[10] Gansner, Emden R. ; Koutsofios, Eleftherios ; North, Stephen C. ; Vo, Kiem-Phong: A Technique for Drawing Directed Graphs. In: *IEEE Trans. Software Eng.* 19 (1993), S. 214–230

[11] Ivanov, Deyan: *Interactive Visualization of Verification Results from CPAchecker with D3*. Bachelor's Thesis, LMU Munich, Software Systems Lab, 2017

[12] Beyer, Dirk ; Keremoglu, M. E. ; Wendler, Philipp: Predicate Abstraction with Adjustable-Block Encoding. In: *Proceedings of the 10th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2010, Lugano, October 20-23)*, FMCAD, 2010, 189-197

A.1 Survey Content

The questionnaire created with Google Forms is attached as pdf file on the following pages.

Evaluation of a web frontend for visualization of computation steps and their results

My bachelor thesis provides the design and implementation of a new web frontend for visualization of computation steps and their results of the software verification tool CPAchecker. The aim of this survey is to
evaluate the use of this web frontend as a tool for students to compare and correct their exercise results of manual creation of an abstract reachability graph (ARG) using the CPA (Configurable Program Analysis) algorithm, and as a tool for the general user to improve his understanding of the analysis results.

Your precious time should be a good investment in improving the new web frontend. Please give me about 30 min of your time to fill out this survey. The tasks should help you to discover and test the features. Maybe you have some great suggestions for improving the new web frontend which you would like to share with me.
Have fun!

General Questions

To find out how well you know the CPAchecker

1. Are you familiar with CPAchecker?

Markieren Sie nur ein Oval.

	1	2	3	4	5	
Yes, I know it by heart	◯	◯	◯	◯	◯	No

2. How often do you use the report.html provided by CPAchecker?

Markieren Sie nur ein Oval.

- ◯ Daily
- ◯ Weekly
- ◯ Monthly
- ◯ Less than monthly
- ◯ Never

Report features

I would like to know how regularly you use them

3. How often do you use the CFA-tab?

Markieren Sie nur ein Oval.

	1	2	3	4	5	
Every time	◯	◯	◯	◯	◯	Never

4. How often do you click on edges of the CFA to switch to the source code line?

Markieren Sie nur ein Oval.

	1	2	3	4	5	
Very often	◯	◯	◯	◯	◯	Only by accident

5. How often do you use the ARG-tab?

Markieren Sie nur ein Oval.

	1	2	3	4	5	
Every time	◯	◯	◯	◯	◯	Never

6. How often do you click on elements of the ARG?

Markieren Sie nur ein Oval.

	1	2	3	4	5	
Very often	◯	◯	◯	◯	◯	Only by accident

7. How often do you use the source-tab?

Markieren Sie nur ein Oval.

	1	2	3	4	5	
Every time	◯	◯	◯	◯	◯	Never

8. How often do you use the log-tab?

Markieren Sie nur ein Oval.

	1	2	3	4	5	
Every time	◯	◯	◯	◯	◯	Never

9. How often do you use the statistics-tab?

Markieren Sie nur ein Oval.

	1	2	3	4	5	
Every time	◯	◯	◯	◯	◯	Never

10. How often do you use the configurations-tab?

Markieren Sie nur ein Oval.

	1	2	3	4	5	
Every time	◯	◯	◯	◯	◯	Never

Output folder

In the output folder you will also find results of the verification run in other formats

11. Have you ever used the ARG.dot file in the output folder?

Markieren Sie nur ein Oval.

- ◯ Yes, regularly
- ◯ Yes, once or rarely
- ◯ No

Task 1

This task is intended to get familiar with the new web frontend and its functionalities. Based on your user experience the following questions shall give me information about the operability and hopefully ideas for improvements.

1. Please read carefully the CPAchecker Documentation "Getting started with CPAchecker" (https://cpachecker.sosy-lab.org/doc.php).

2. Please checkout the CPAchecker branch from svn with "svn checkout https://svn.sosy-lab.org/software/cpachecker/branches/SonjaM/ cpachecker".

3. Run the analysis with configuration "-setprop solver.solver=smtinterpol -setprop cpa.predicate.encodeBitvectorAs=integer -setprop cpa.predicate.encodeFloatAs=rational -collector doc/examples/example.c".

4. You will find the ComputationSteps.html in the output folder, open it with any browser.

5. Please make yourself familiar with the different features of the web frontend. Note the Source Code and the Computation Steps Graph boxes. Try the buttons, slider, splitter and hyperlink (marked on the following screenshot).

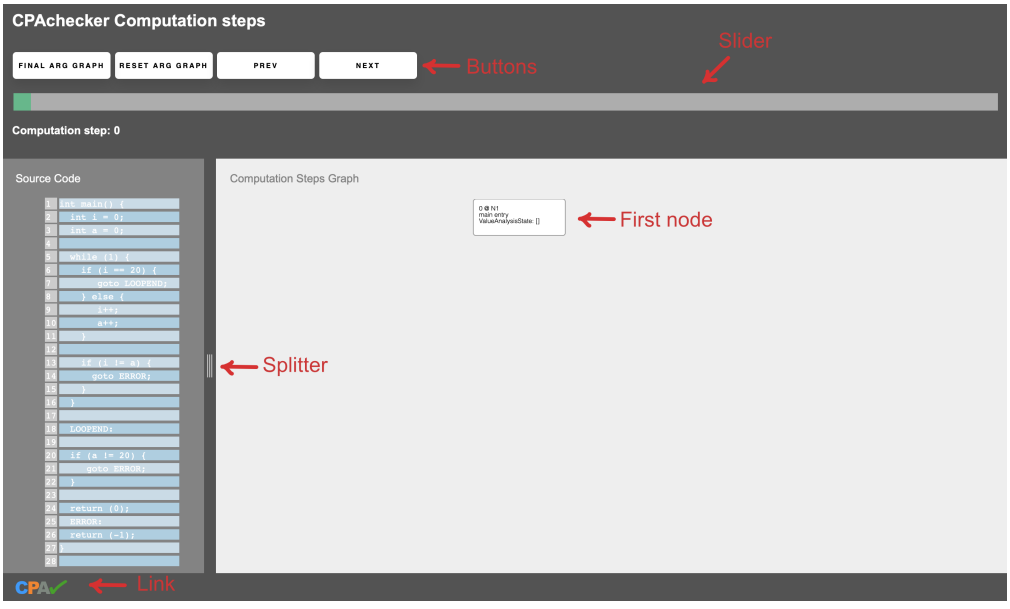
6. Note the drop down button FINAL ARG GRAPH. Select a final ARG and compare it with the other.

7. Build the graph with the slider or NEXT button, try to zoom and pan the graph. Then hover the mouse over nodes and make yourself familiar with the informations in the tooltip box.

8. Press the RESET ARG GRAPH button. Step forward with the NEXT button or the slider and step backwards with the PREV button or the slider. Note the dark blue highlighting of the source code.

9. Move the splitter to the right and left, click the CPA logo.

Screenshot



12. Was it clear where to find the ComputationSteps.html?

Markieren Sie nur ein Oval.

	1	2	3	4	5	
Yes	◯	◯	◯	◯	◯	Not at all

13. Which browser did you use?

_____

14. If there were any difficulties or display problems with your chosen browser, please name them?

_____

_____

_____

_____

_____

15. Did the CPA-logo hyperlink work?

Markieren Sie nur ein Oval.

	1	2	3	4	5	
Yes, no problem	◯	◯	◯	◯	◯	No

16. Could you slide the Source Code and Computation Steps Graph boxes with the splitter to your desired width?

Markieren Sie nur ein Oval.

	1	2	3	4	5	
Yes	◯	◯	◯	◯	◯	Not at all

17. Have you noticed that the highlighted source code line (dark blue) matches the edge label?

Markieren Sie nur ein Oval.

	1	2	3	4	5	
Yes, perfectly obvious	◯	◯	◯	◯	◯	Not noticed

18. Was the graph displayed in the Computation Steps Graph box?

Markieren Sie nur ein Oval.

	1	2	3	4	5	
Yes, from the first to the last node	◯	◯	◯	◯	◯	Not at all

19. How did the graph zooming work?

Markieren Sie nur ein Oval.

	1	2	3	4	5	
Flawless	◯	◯	◯	◯	◯	Not at all

20. Were you able to pan/drag-and-drop the graph easily?

Markieren Sie nur ein Oval.

	1	2	3	4	5	
Yes, easy	◯	◯	◯	◯	◯	Not at all

21. Did the mouse over showed the tooltip box?

Markieren Sie nur ein Oval.

- ◯ Yes
- ◯ No

22. Was the content of the tooltip box clear to you?

Markieren Sie nur ein Oval.

	1	2	3	4	5	
Yes	◯	◯	◯	◯	◯	No

23. Do you have any suggestions what could be additionally displayed in the tooltip box?

24. How was the performance of the slider?

Markieren Sie nur ein Oval.

	1	2	3	4	5	
Good	◯	◯	◯	◯	◯	Bad

25. Did the NEXT button worked properly?

Markieren Sie nur ein Oval.

	1	2	3	4	5	
Yes, from the first to the last node	◯	◯	◯	◯	◯	Not at all

26. Did the PREV button worked proberly?

Markieren Sie nur ein Oval.

	1	2	3	4	5	
Yes from the last to the first step	◯	◯	◯	◯	◯	Not at all

27. When pressing the reset button, was the graph deleted and the first node displayed again?

Markieren Sie nur ein Oval.

- ◯ Yes
- ◯ No
- ◯ There were problems loading the page
- ◯ The slider and/or the Computation Step Display was not reseted

28. Did the drop down menu FINAL ARG GRAPH worked for both options?

Wählen Sie alle zutreffenden Antworten aus.

- ☐ Yes
- ☐ Only the HIGHLIGHT MERGE option resulted in displaying the graph
- ☐ Only the STANDARD ARG option resulted in displaying the graph
- ☐ No

Task 2

This task is to find out if the new web frontend is suitable to make the calculation of the merge operator of the CPA easier to understand for students.

1.Please have a closer look at following figures. They show an example C-program to be analyzed and its representation as CFA and the CPA algorithm, the CPAchecker uses.

2. Run the CPAchecker analysis with configuration "-setprop solver.solver=smtinterpol -setprop cpa.predicate.encodeBitvectorAs=integer -setprop cpa.predicate.encodeFloatAs=rational -collector doc/examples/exampleSM.c".

3. Open the ComputationSteps.html and choose the STANDARD ARG in the drop down menu FINAL ARG GRAPH,
then zoom and pan until you find the node 14@N5 with ARG ID 14.
Then choose the HIGHLIGHT MERGE in the drop down menu FINAL ARG GRAPH, zoom and pan again until you find the node 14@N5 with ARG ID 14. Note the difference.
Performing the CPA algorithm on the provided C-program, represented by the CFA, results in a merge at node N5 of the CFA (see CFA figure).
Note that the standard ARG in your browser shows the merged node (14@N5) with ValueAnalysisState (x = 1).
A main advantage of the new web frontend is the display of the calculation steps of the merge operator. The merge operator is defined as merge(e',e")= e". Note that the merge partners (e') will not be displayed in the final standard ARG.

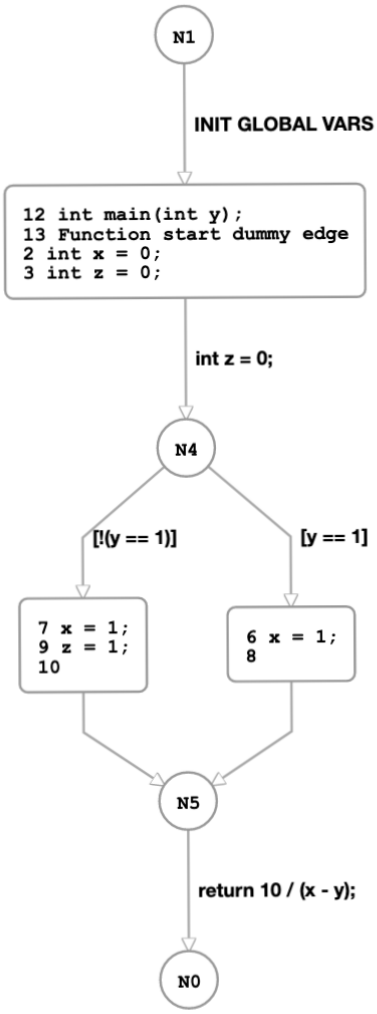
4. Please press the RESET ARG GRAPH button. Now follow the graph evolution carefully by clicking on the NEXT button or using the slider until you have reached Computation Step 13 or node 13@N5.

5. A typical exercise for student is:
"Perform a CPA analysis on the given program (represented by the CFA) and represent the resulting set of reachable states as an abstract reachability graph (ARG)."
To solve this exercise you need to calculate the merge manually. Before you are able to draw the final ARG,
you had to calculate the states which will get merged.
Try to understand why node 13@N5 has the ValueAnalysisState(x=1, y=1, z=0) and node 10@N5 has the ValueAnalysisState(x=1, z=01). These are the states e' which will get merged to node 14@N5 with ValueAnalysisState (x = 1). Note the colors of the nodes.

5. Now go to the next calculation step by pressing the NEXT button or using the slider. Watch what happens. If necessary, go back and forth using either the slider or the PREV/NEXT buttons.

C-Program and Control-flow automaton (CFA)

```
1  int main(int y) {
2    int x = 0;
3    int z = 0;
4
5   if (y== 1){
6   x=1;
7   }else{
8   x=1;
9   z=1;
10  }
11  return 10/ (x-y);
12 }
```



CPA Algorithm

Algorithm 1. $CPA(\mathbb{D}, e_0)$

Input: a configurable program analysis $\mathbb{D} = (D, \rightsquigarrow, \text{merge}, \text{stop})$,
an initial abstract state $e_0 \in E$, let E denote the set of elements of the semi-lattice of D
Output: a set of reachable abstract states
Variables: a set reached of elements of E, a set waitlist of elements of E

```
waitlist := {e_0}
reached := {e_0}
while waitlist ≠ ∅ do
  pop e from waitlist
  for each e' with e⇝e' do
    for each e'' ∈ reached do
      // Combine with existing abstract state.
      e_new := merge(e', e'')
      if e_new ≠ e'' then
        waitlist := (waitlist ∪ {e_new}) \ {e''}
        reached := (reached ∪ {e_new}) \ {e''}
    if ¬ stop(e', reached) then
      waitlist := waitlist ∪ {e'}
      reached := reached ∪ {e'}
return reached
```

29. Was it easy to find node 14@N5?

Markieren Sie nur ein Oval.

	1	2	3	4	5	
Yes, super easy	◯	◯	◯	◯	◯	No, I almost surrendered

30. Was it clear how to get to computation step 13?

Markieren Sie nur ein Oval.

	1	2	3	4	5	
Sure	◯	◯	◯	◯	◯	No, I looked for the node 13@N5

31. Does the colors of states e' which will get merged and the merged state e" help to unterstand the merge operation?

Markieren Sie nur ein Oval.

	1	2	3	4	5	
Yes, great help	◯	◯	◯	◯	◯	Completely superfluous and confusing

32. Was it clear that node 14@N5 is the merged state of 13@N5 and 10@N5?

Markieren Sie nur ein Oval.

	1	2	3	4	5	
Yes	◯	◯	◯	◯	◯	No

33. Which final ARG do you think a student will use to correct/compare his exercise results?

Wählen Sie alle zutreffenden Antworten aus.

- [ ] Highlight Merge
- [ ] Standard ARG

34. Does the final ARG "Highlight Merge" made the computation steps of the CPA Algorithm clearer?

Markieren Sie nur ein Oval.

	1	2	3	4	5	
Yes, absolutly	◯	◯	◯	◯	◯	Not at all

35. Does the possibility to go back and forth by using PREV/NEXT buttons respectively the slider help to understand computation steps of the CPA Algorithm and therefore the evolution of the ARG?

Markieren Sie nur ein Oval.

	1	2	3	4	5	
Absolutly	◯	◯	◯	◯	◯	Still no clue

36. Does highlighting the source code line help to understand the step-by-step evolution of the ARG?

Markieren Sie nur ein Oval.

	1	2	3	4	5	
Yes a lot	◯	◯	◯	◯	◯	Not at all

37. Do you think the new web frontend is a useful tool for students?

Markieren Sie nur ein Oval.

	1	2	3	4	5	
Yes	◯	◯	◯	◯	◯	No

38. Do you think the new web frontend is useful for the general CPAchecker user?

Markieren Sie nur ein Oval.

	1	2	3	4	5	
Yes	◯	◯	◯	◯	◯	No

General Rating

You are almost ready.....

39. Please rate the usability of the ComputationSteps.html

Markieren Sie nur ein Oval.

	1	2	3	4	5	
Completely unclear where to find and click	◯	◯	◯	◯	◯	Everything was self-explaining

40. 5.2 Please rate the web design

Markieren Sie nur ein Oval.

	1	2	3	4	5	
not attractive	◯	◯	◯	◯	◯	attractive

41. 5.3 Please rate your user-experience

Markieren Sie nur ein Oval.

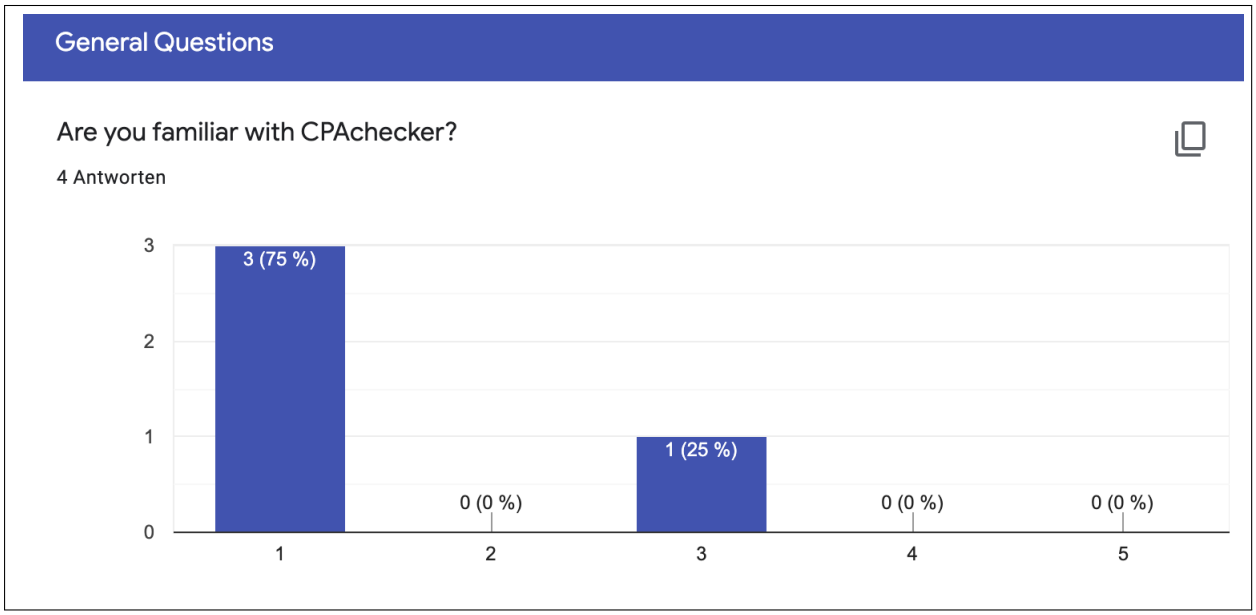
	1	2	3	4	5	
It was a pain working with it	◯	◯	◯	◯	◯	I enjoyed working with it

42. Feel free to express wishes for additional features or make suggestions for improvements to the existing features

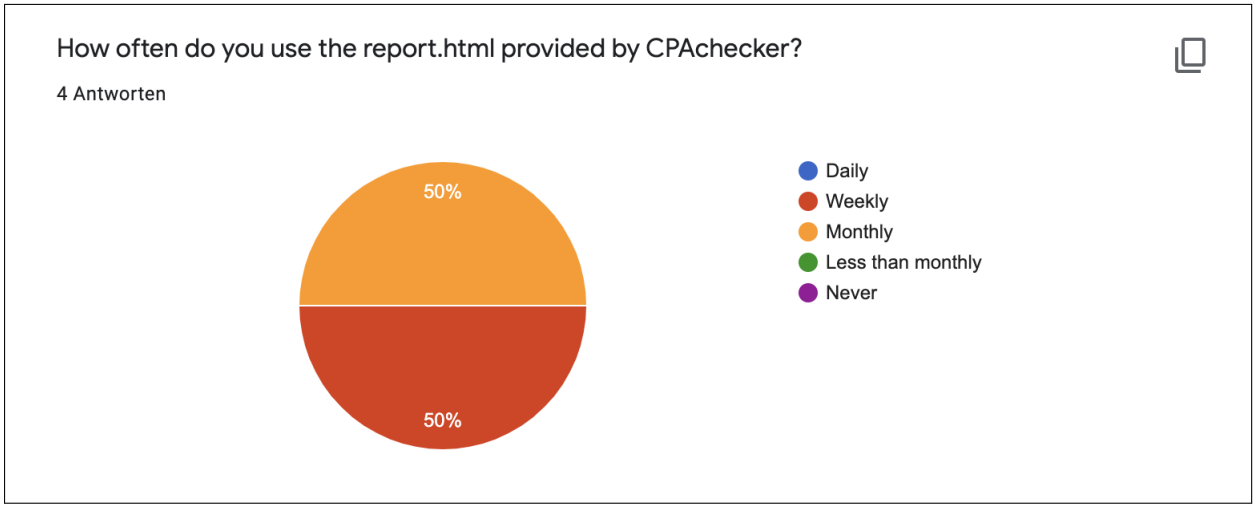
43. Please use this section for additional comments and remarks

THANK YOU

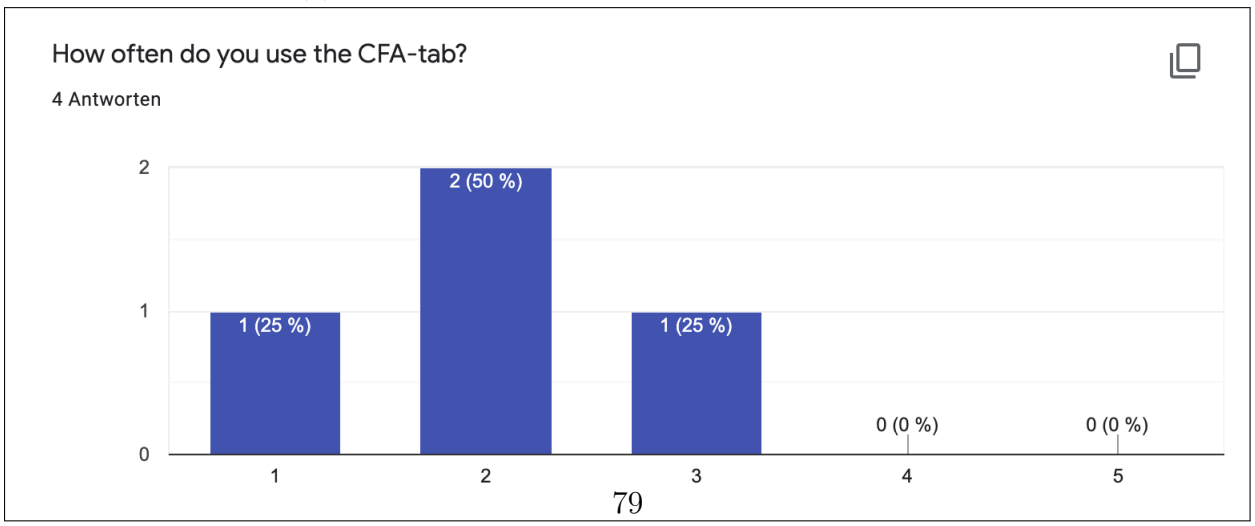
A.2 Survey Results



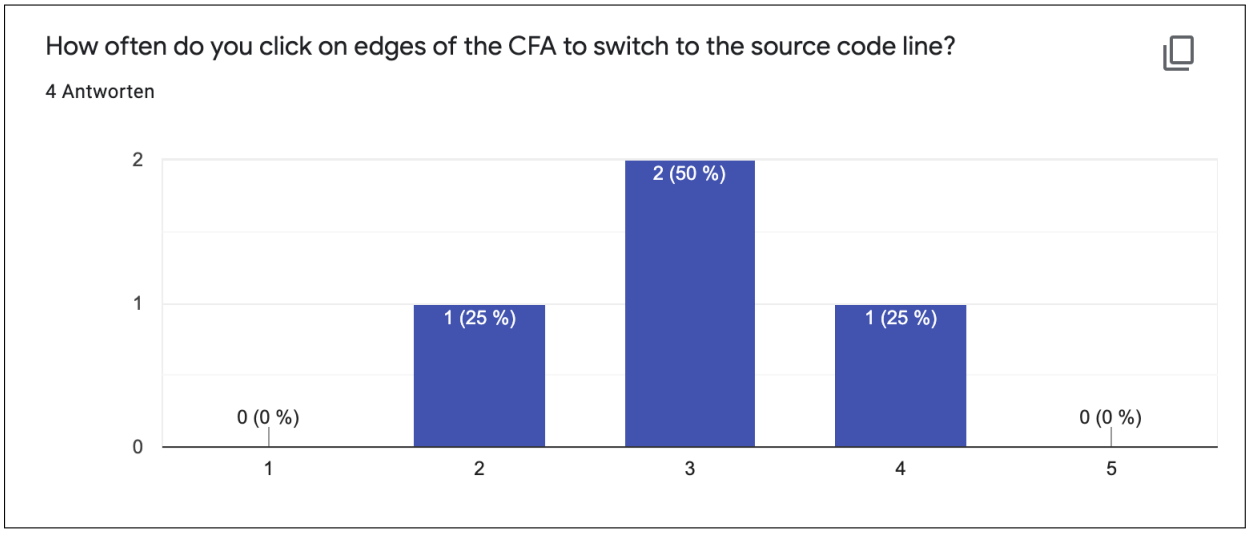
(1) Scale: 1 =Yes,I know it by heart; 5 = No



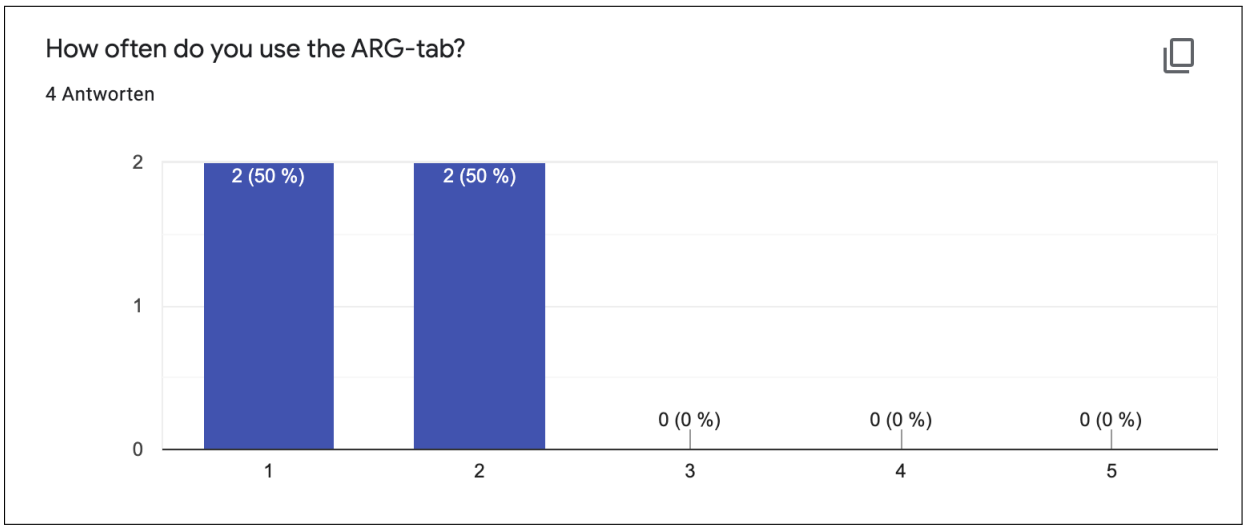
(2)



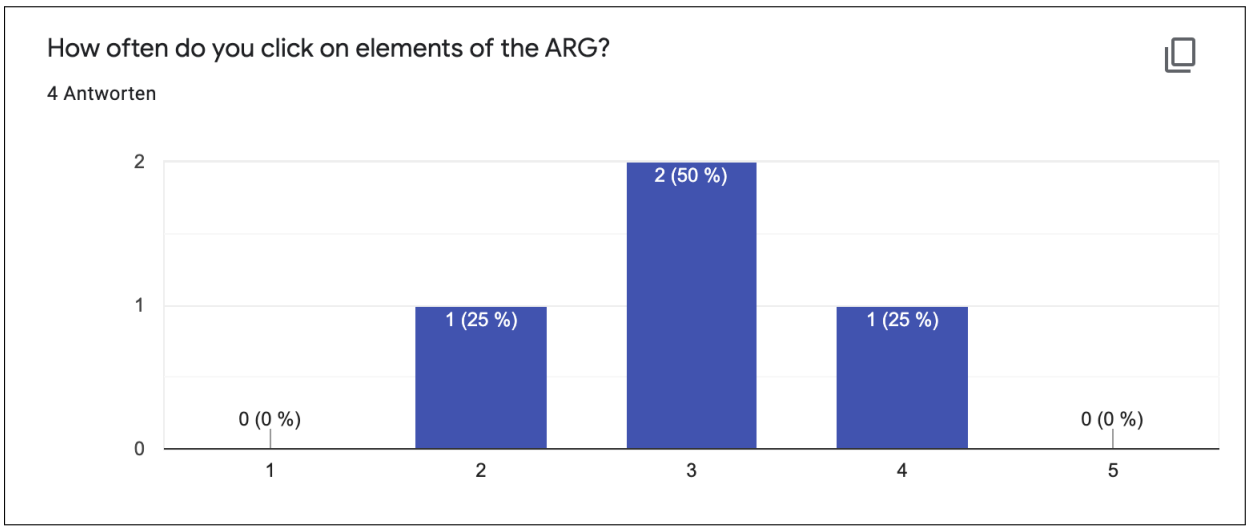
(3) Scale 1 = Every time; 5 = Never



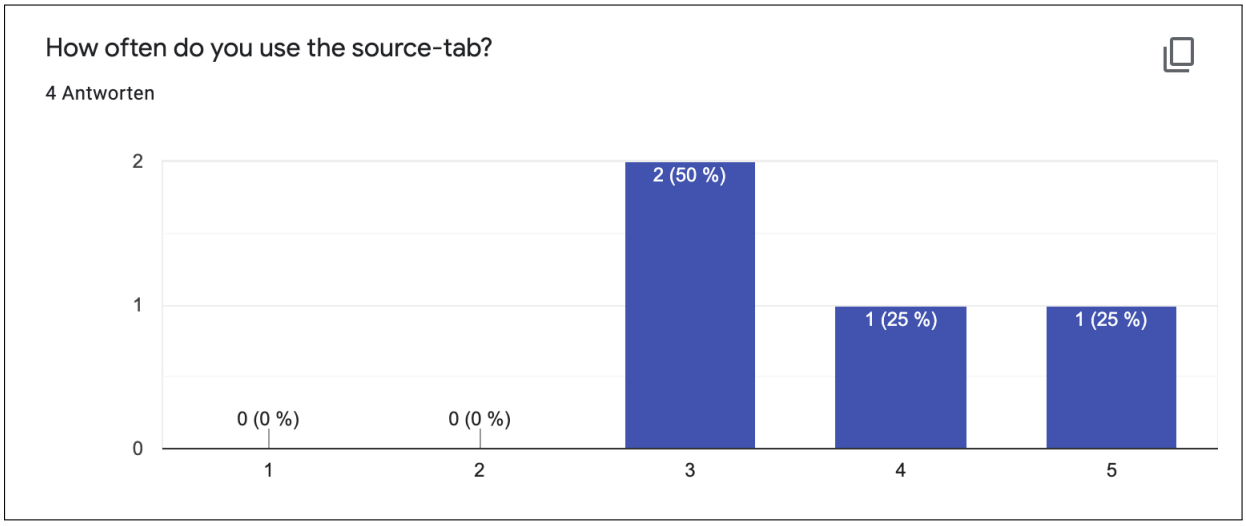
(1) Scale 1 = Very often; 5 = Only by accident



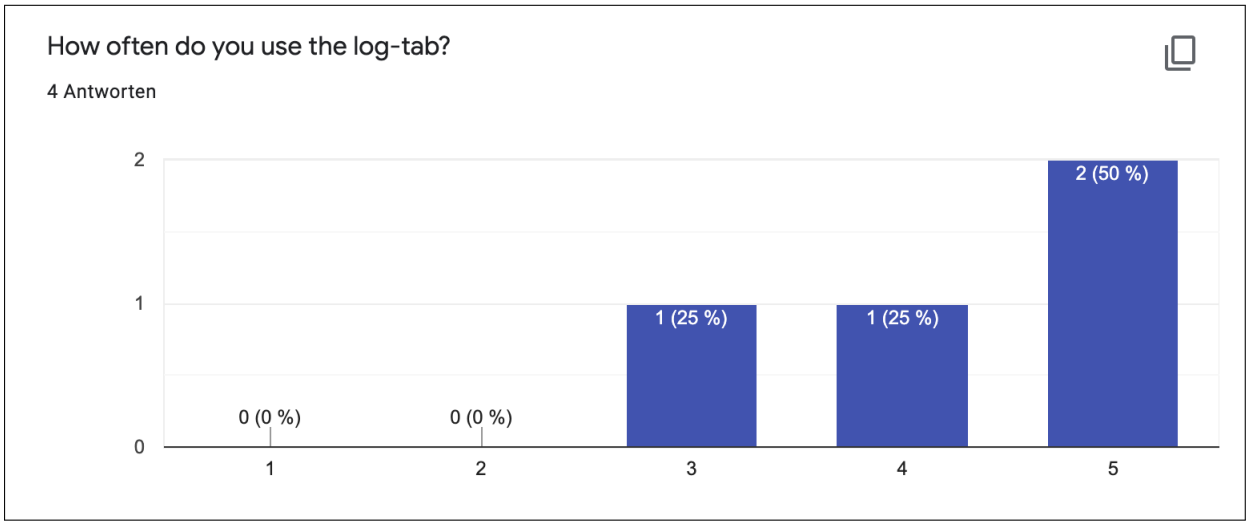
(2) Scale 1 = Every time; 5 = Never



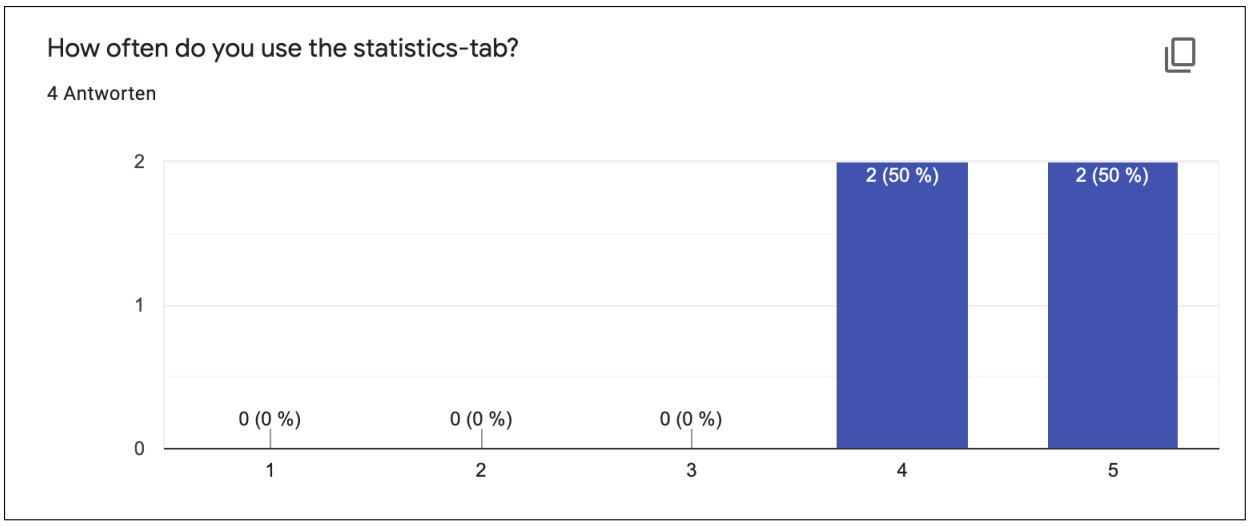
(3) Scale 1 = Very often; 5 = Only by accident



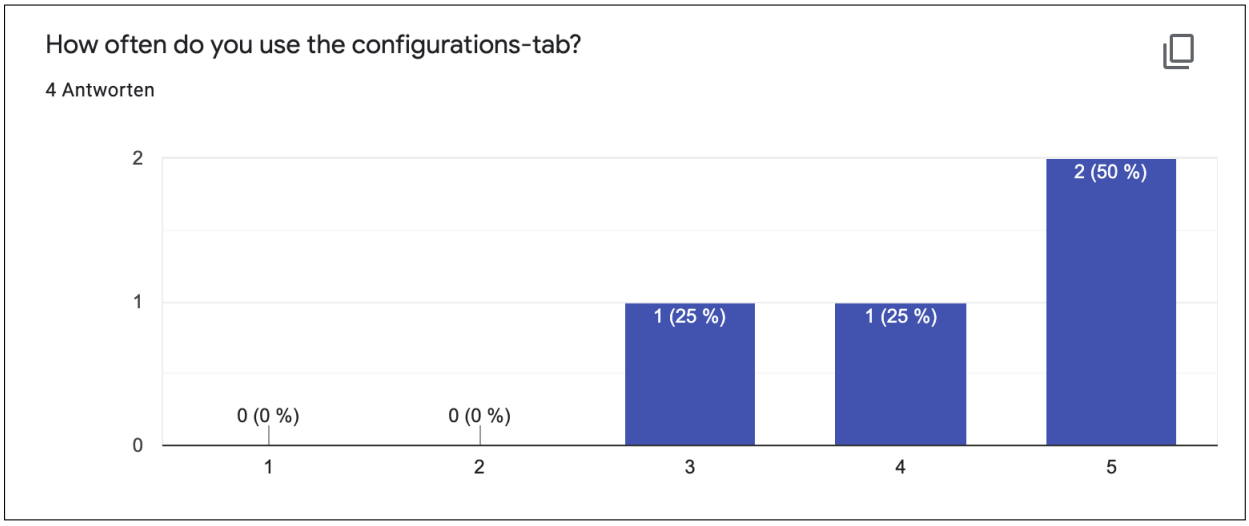
(1) Scale 1 = Every time; 5 = Never



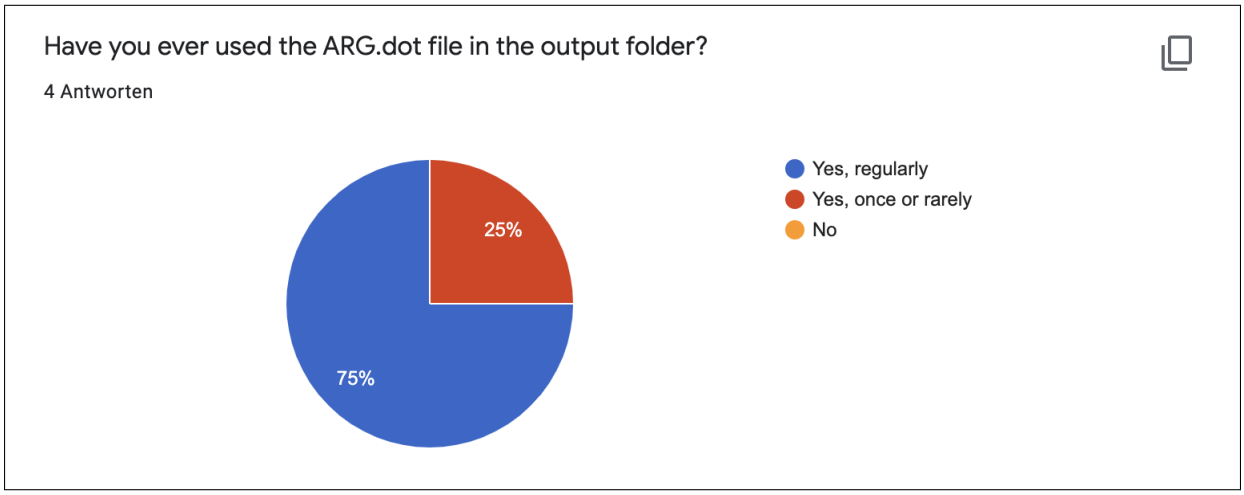
(2) Scale 1 = Every time; 5 = Never



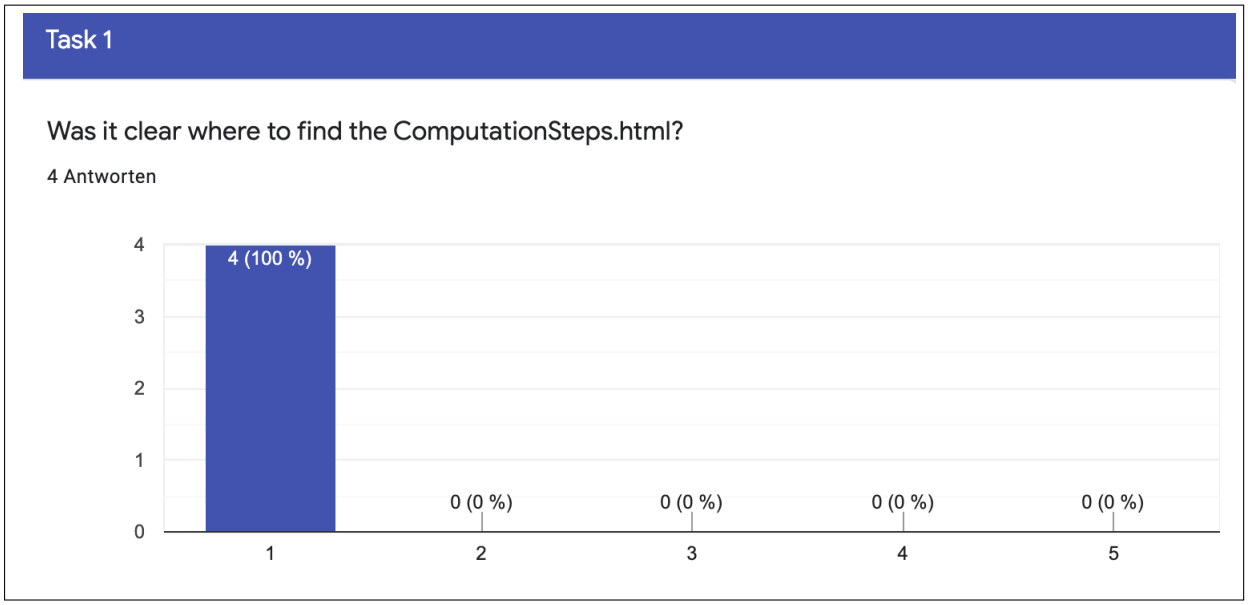
(3) Scale 1 = Every time; 5 = Never



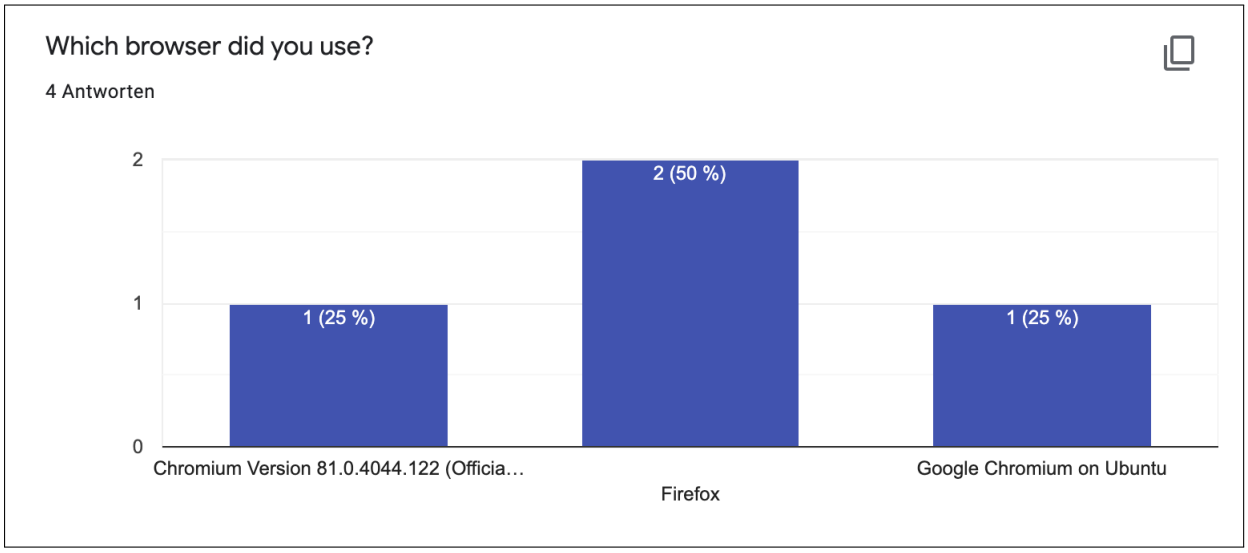
(1) Scale 1 = Every time; 5 = Never



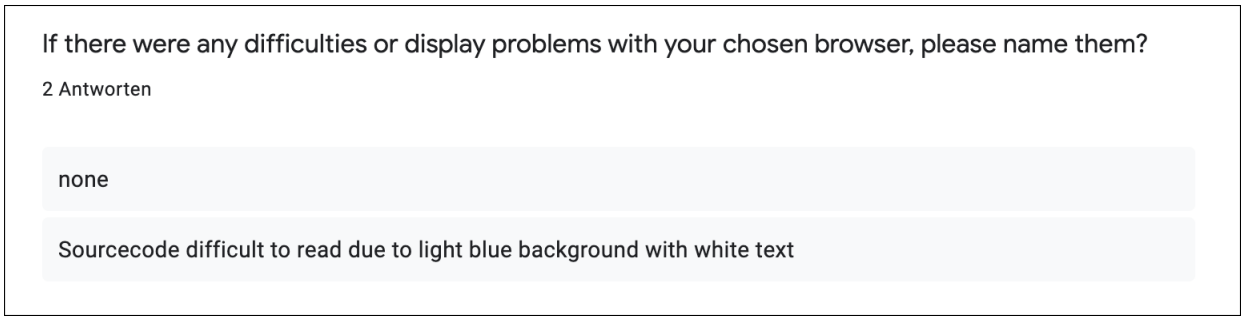
(2)



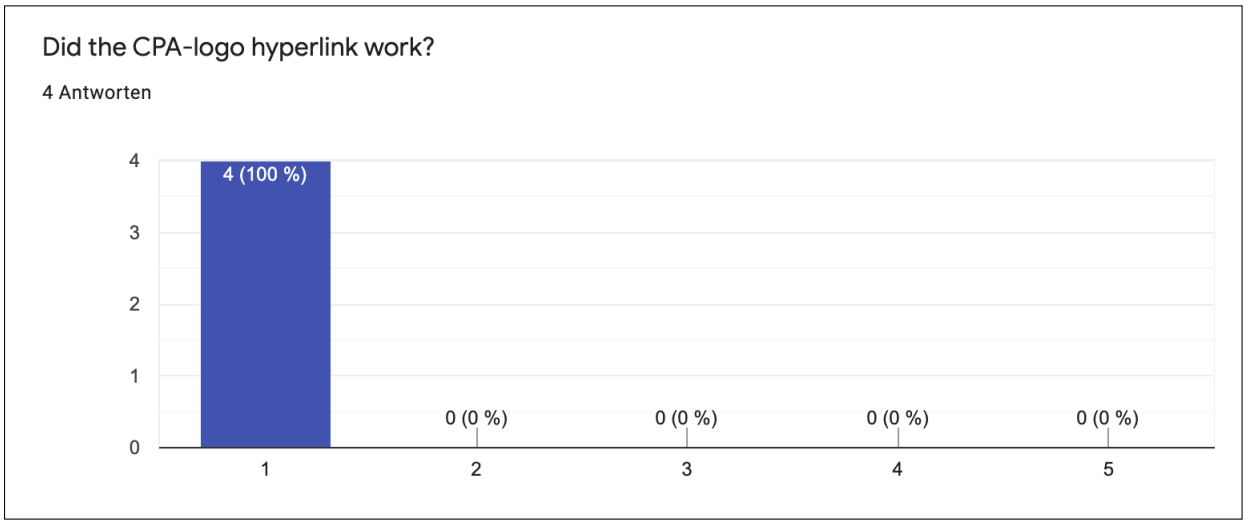
(3) Scale 1 = Yes; 5 = Not at all



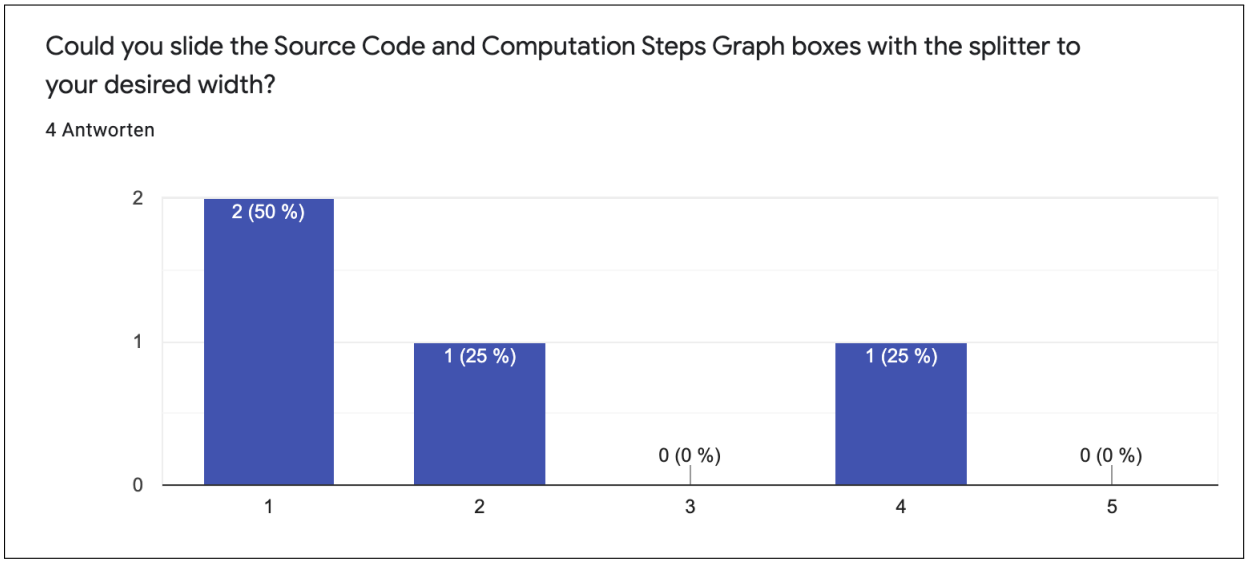
(1)



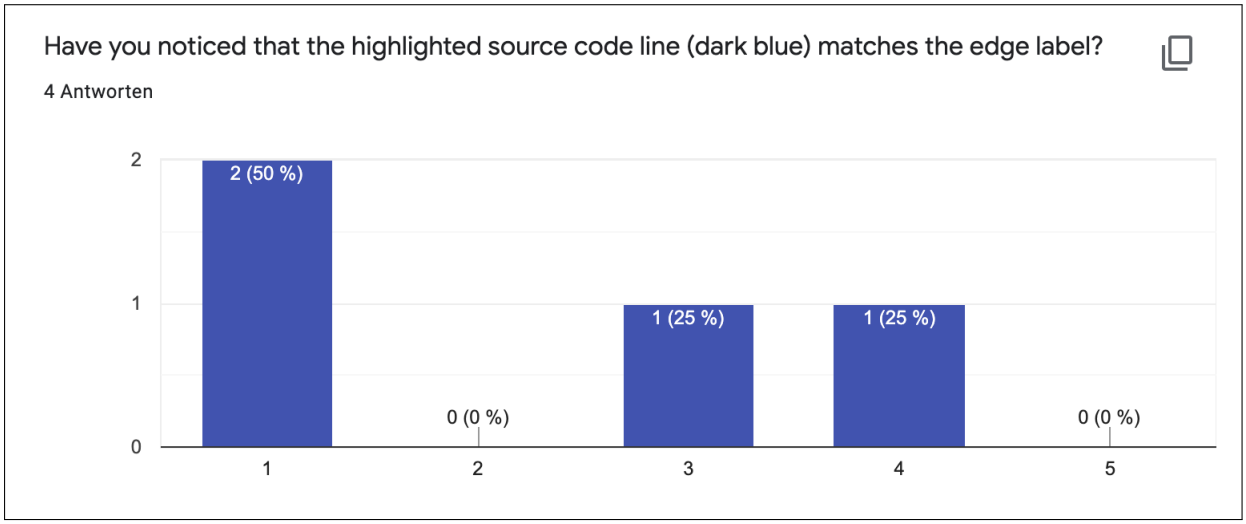
(2)



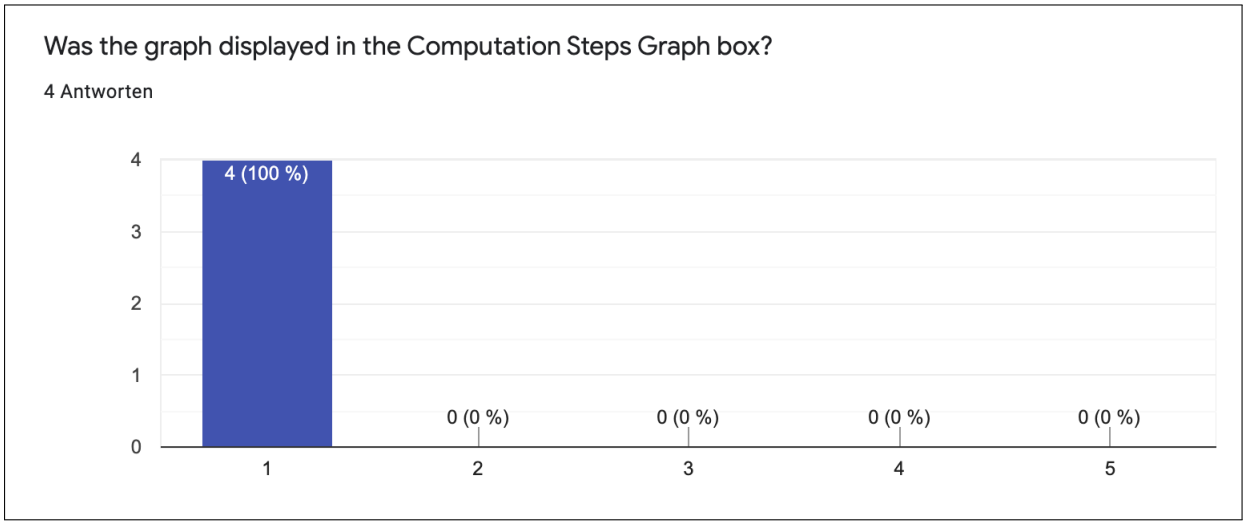
(3) Scale 1 = Yes, no problem; 5 = No



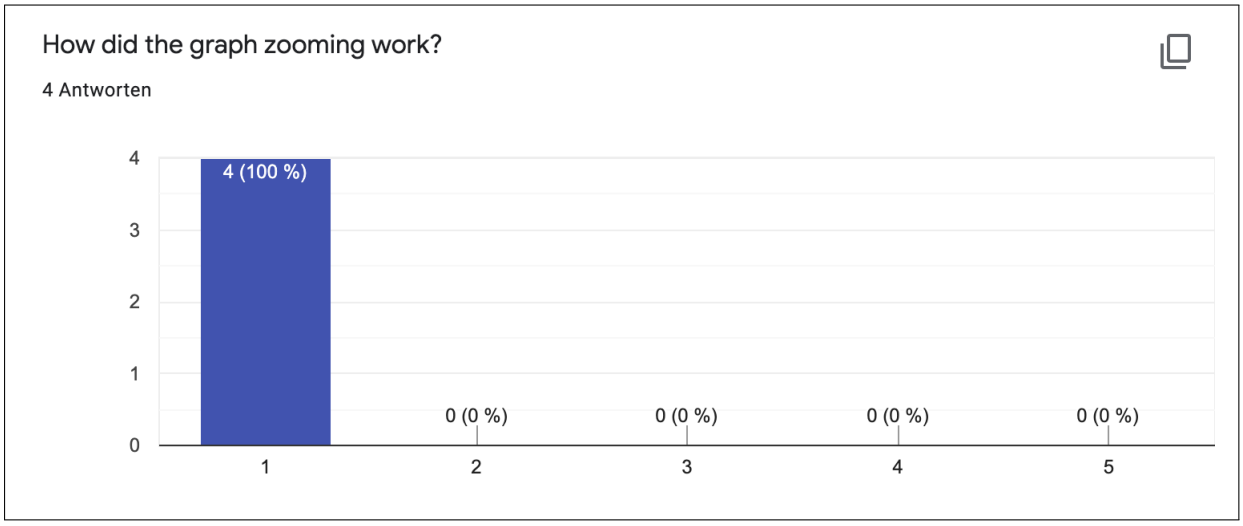
(1) Scale 1 = Yes; 5 = Not at all



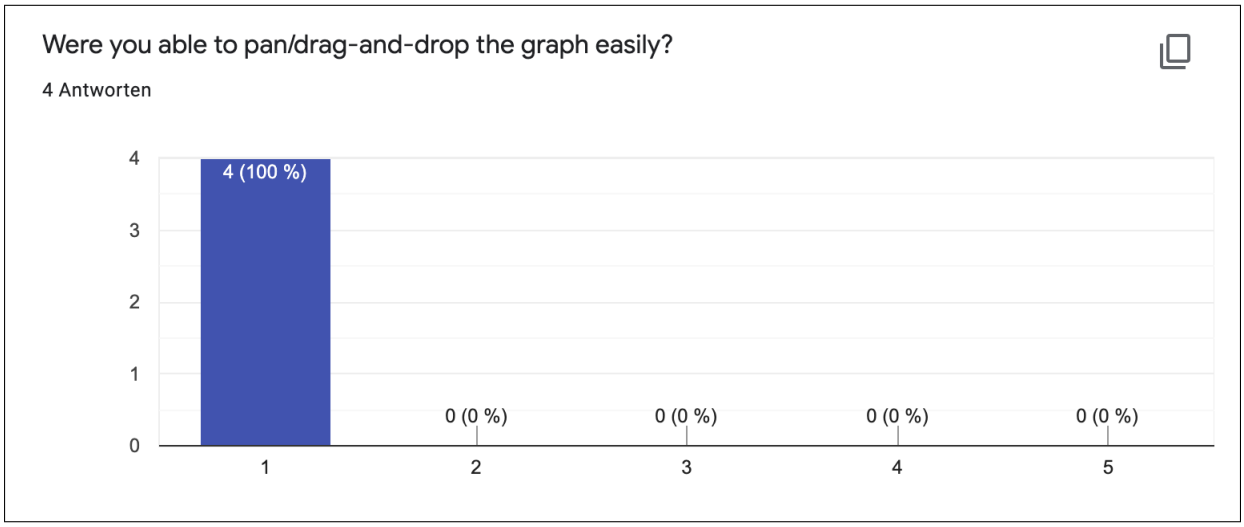
(2) Scale 1 = Yes, perfectly obvious; 5 = Not noticed



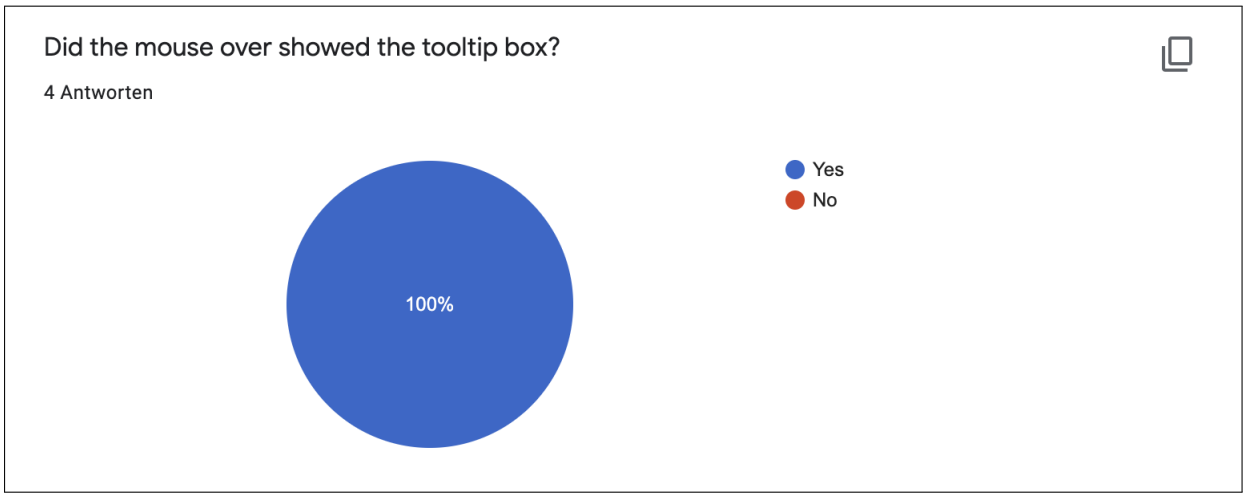
(3) Scale 1 = Yes, from the first to the last node; 5 = Not at all



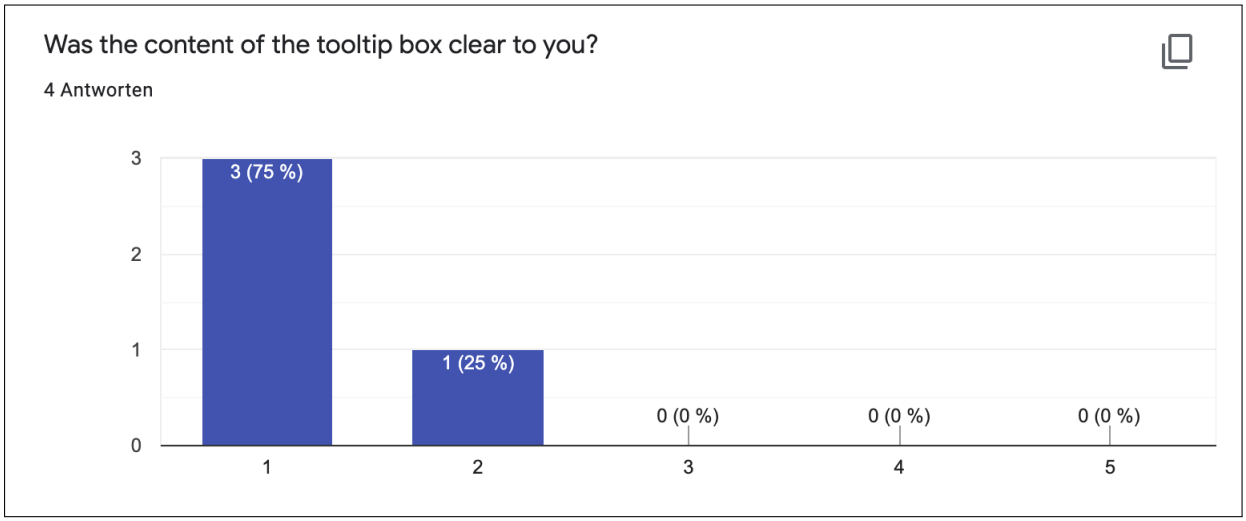
(1) Scale 1 = Flawless; 5 = Not at all



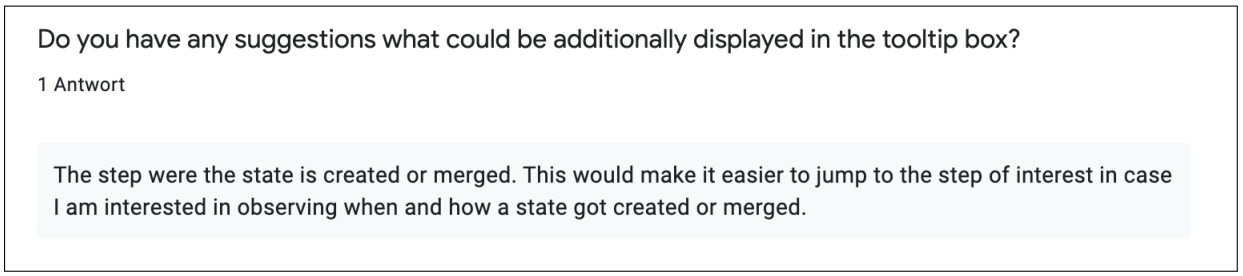
(2) Scale 1 = Yes, easy; 5 = Not at all



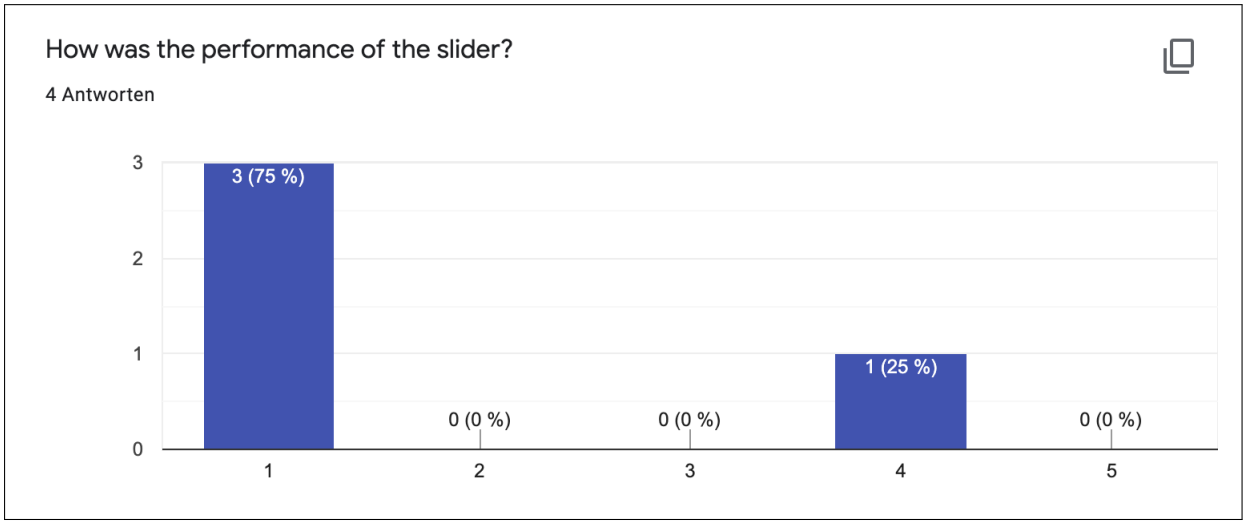
(3)



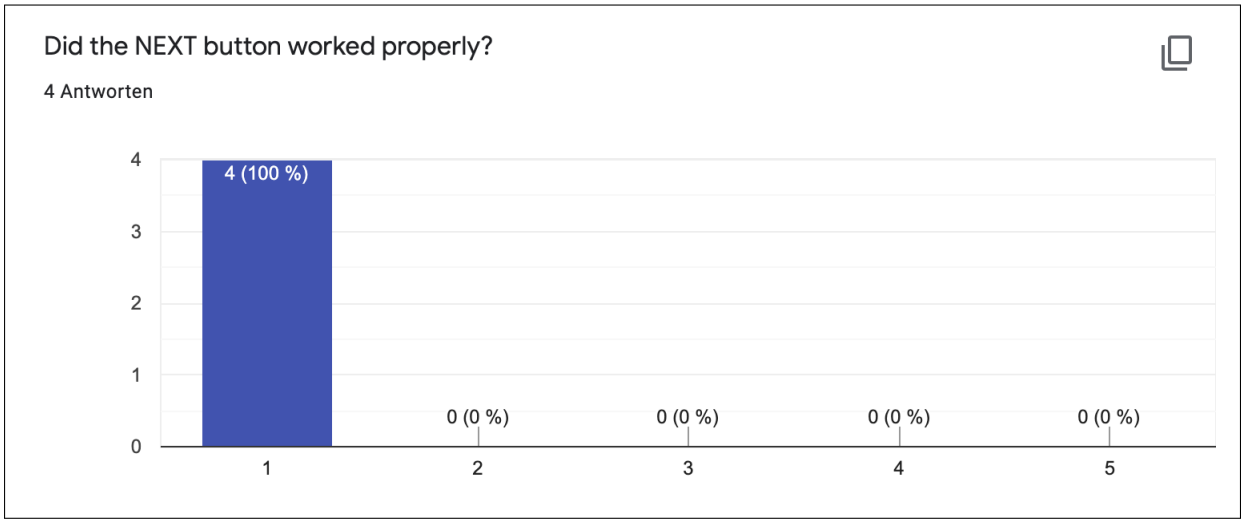
(1) Scale 1 = Yes; 5 = No



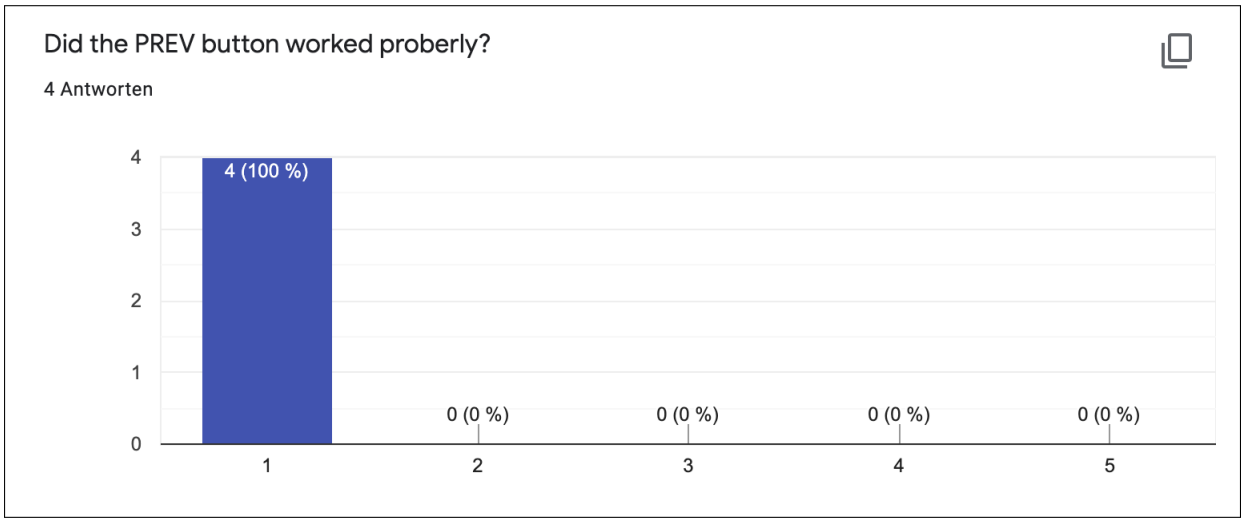
(2)



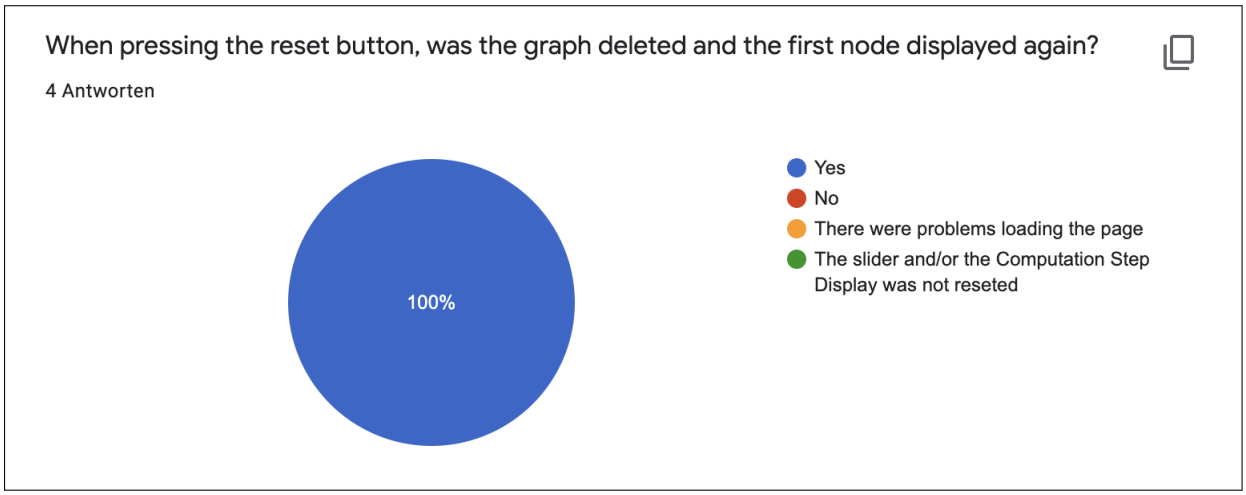
(3) Scale 1 = Good; 5 = Bad



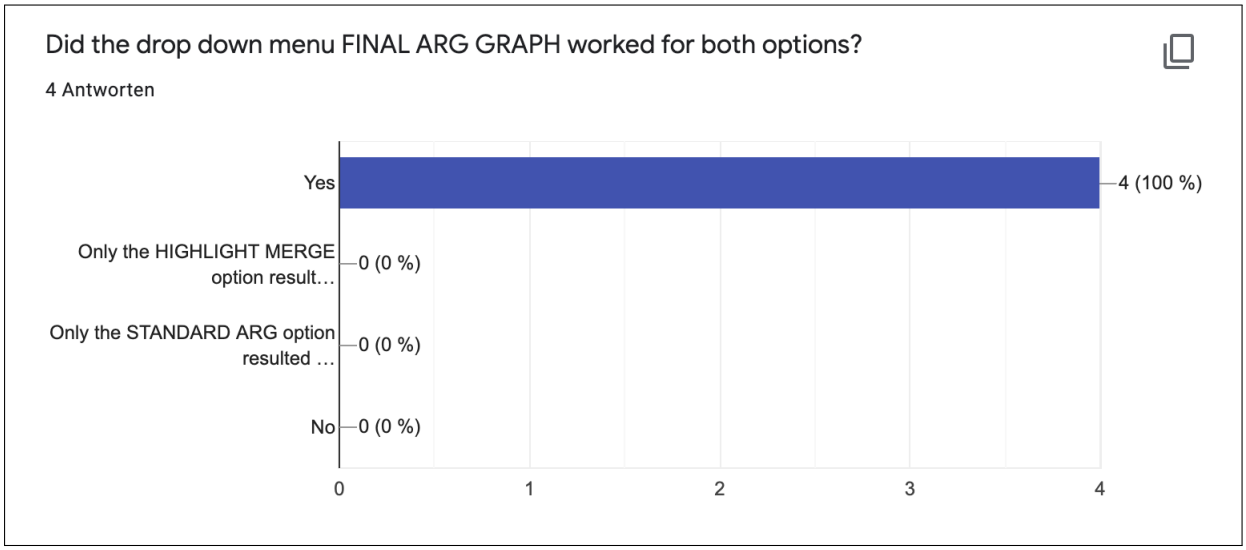
(1) Scale 1 = Yes, from the first to the last node; 5 = Not at all



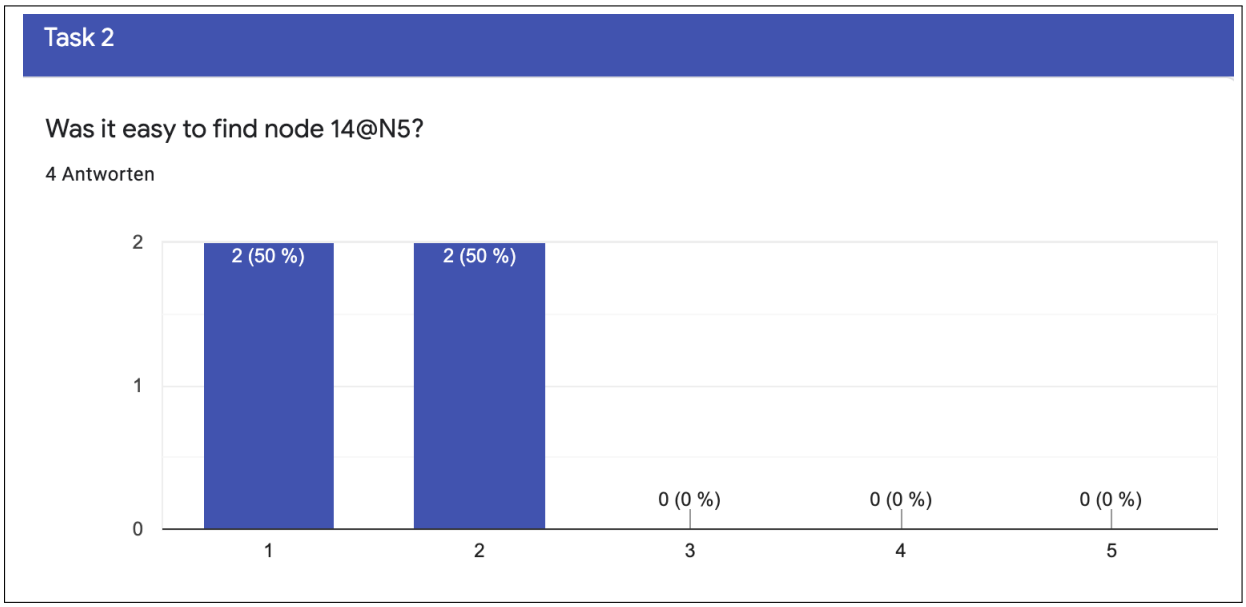
(2) Scale 1 = Yes, from the last to the first step; 5 = Not at all



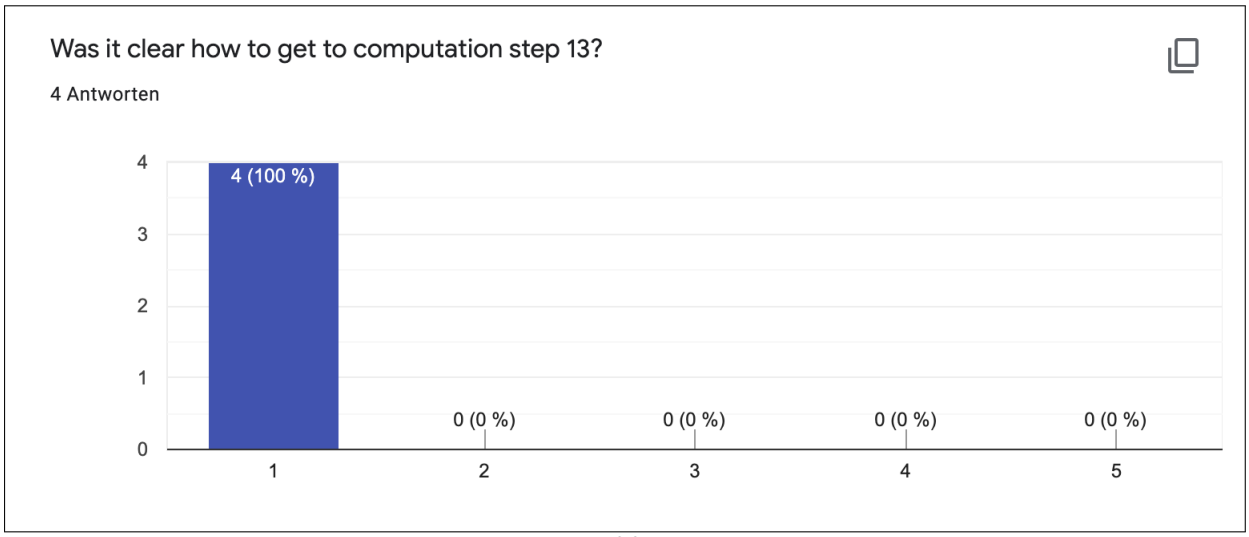
(3)



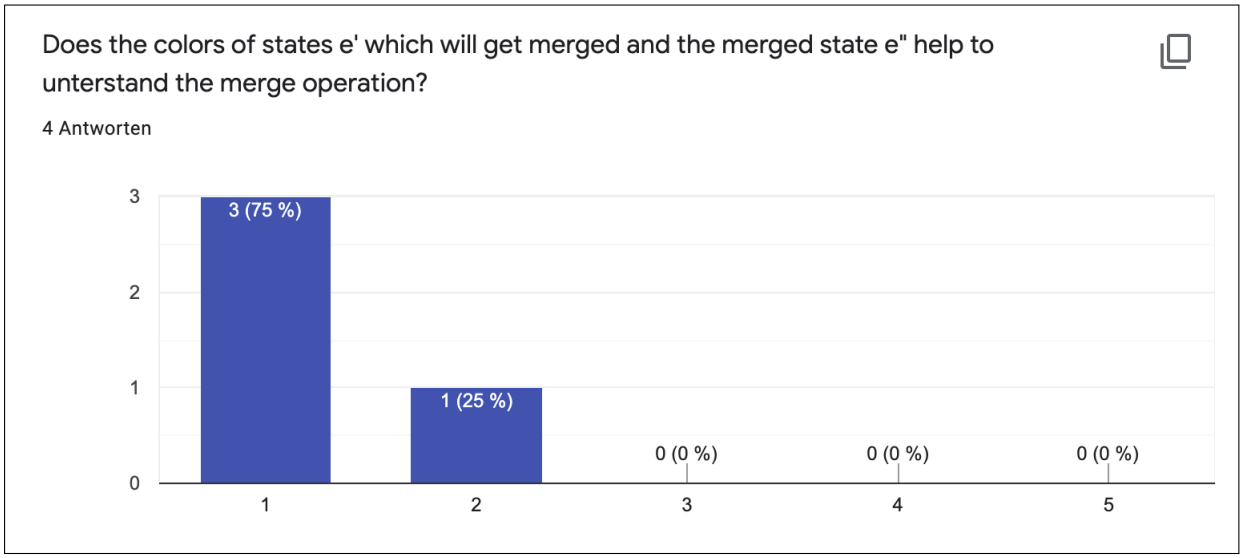
(1)



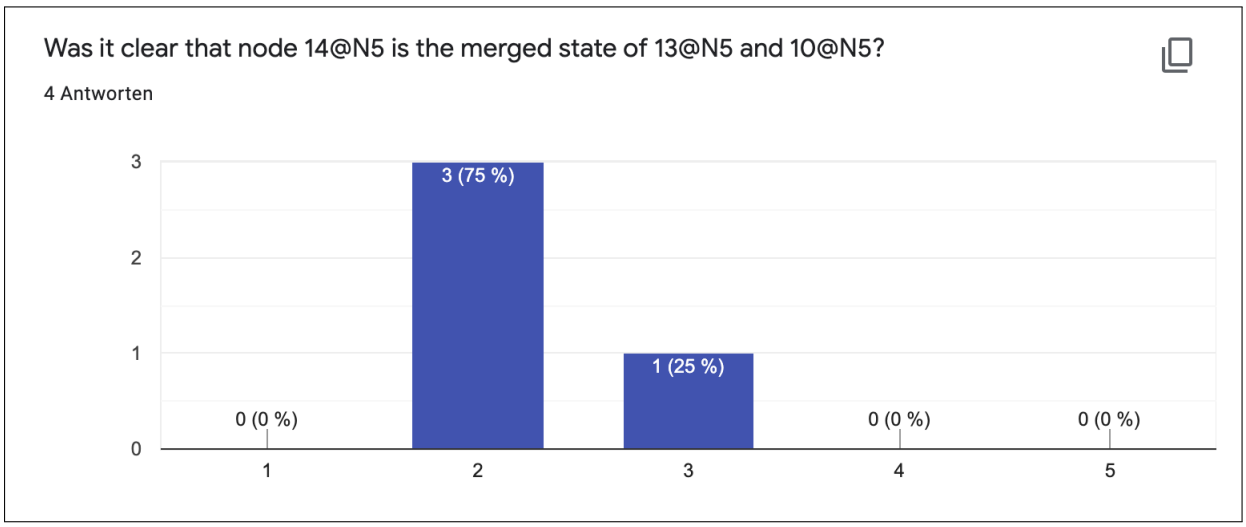
(2) Scale 1 = Yes, super easy; 5 = No, I almost surrendered



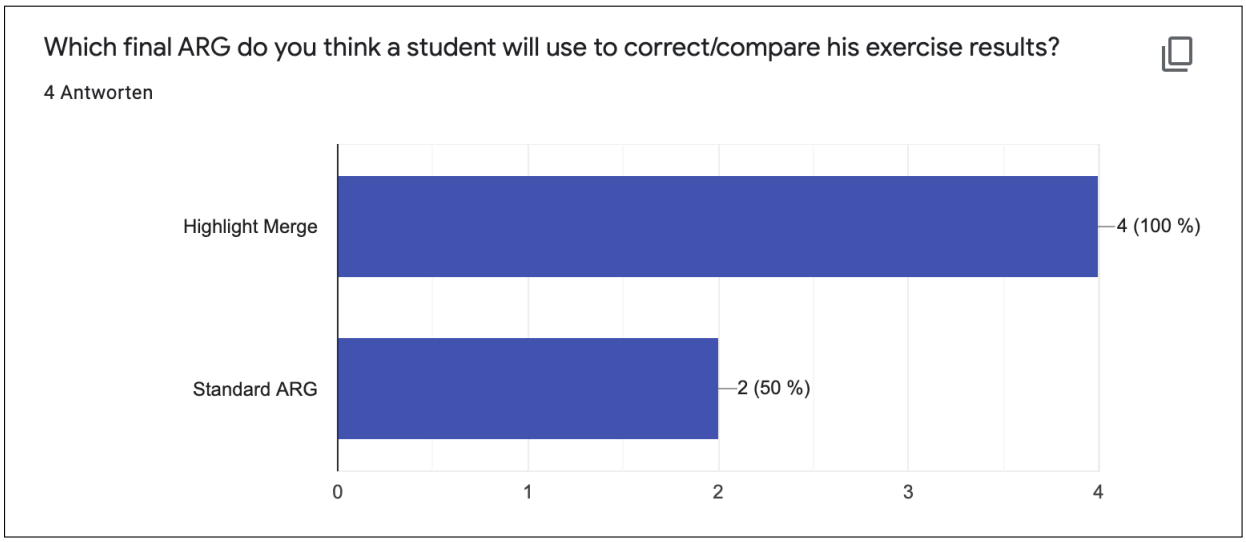
(3) Scale 1 = Sure; 5 = No, I looked for the node 13@N5



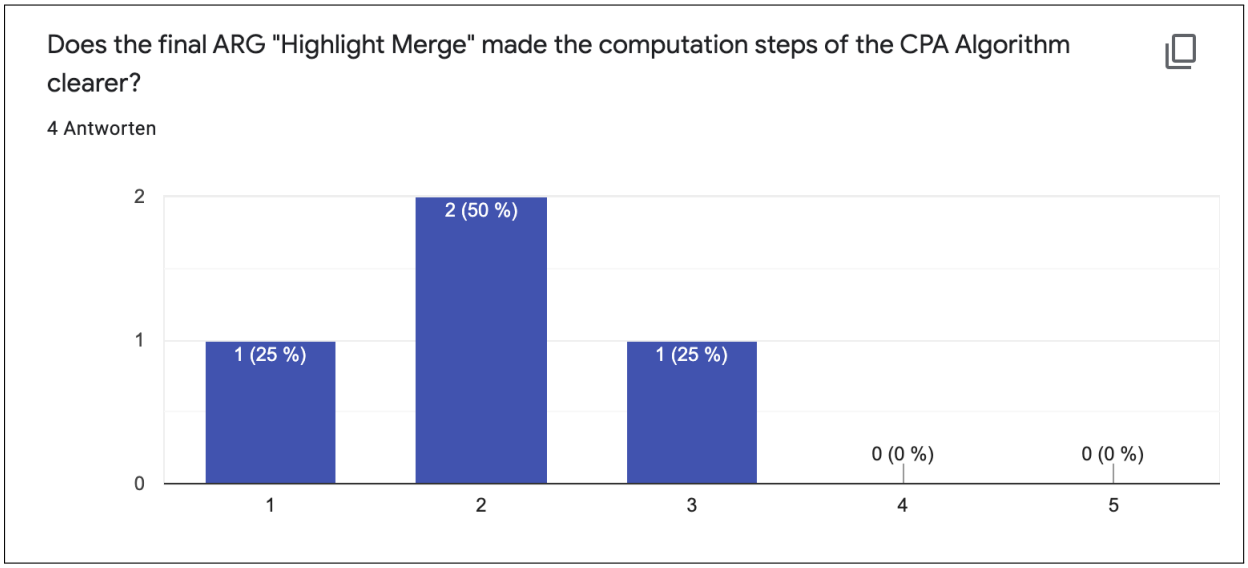
(1) Scale 1 = Yes, great help; 5 = Completely superfluous and confusing



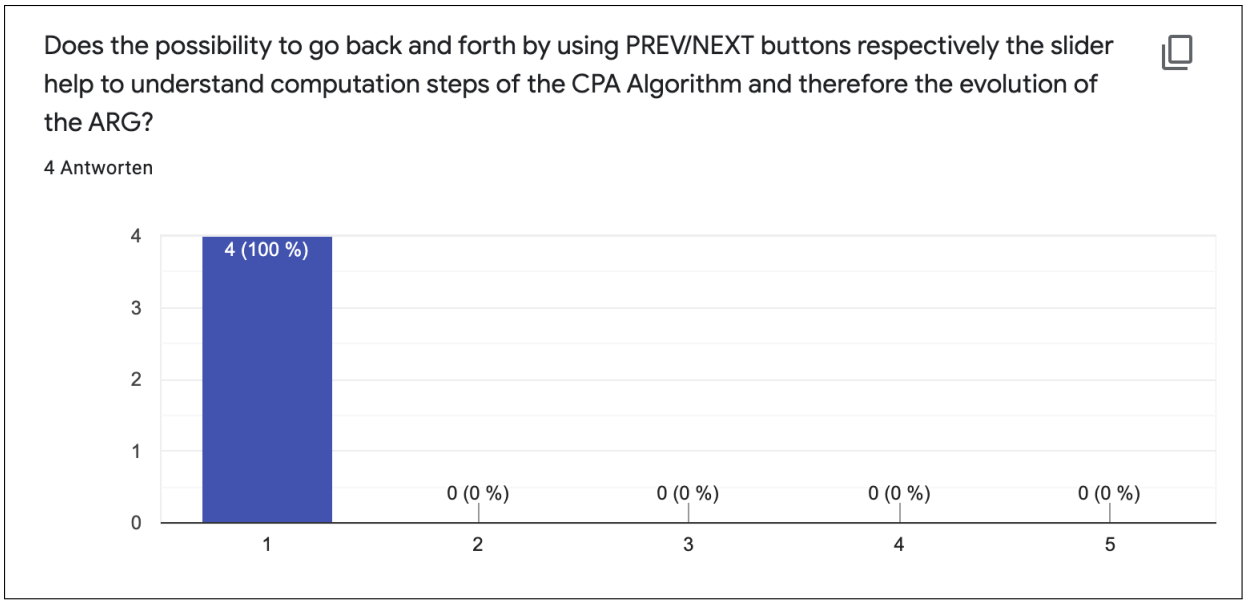
(2) Scale 1 = Yes; 5 = No



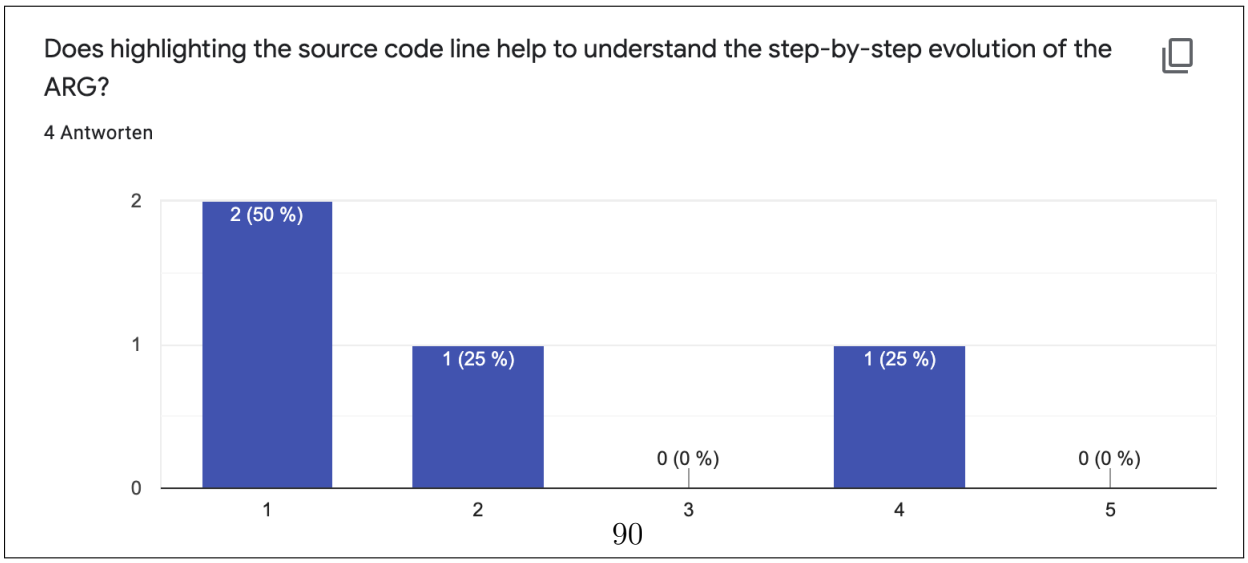
(3)



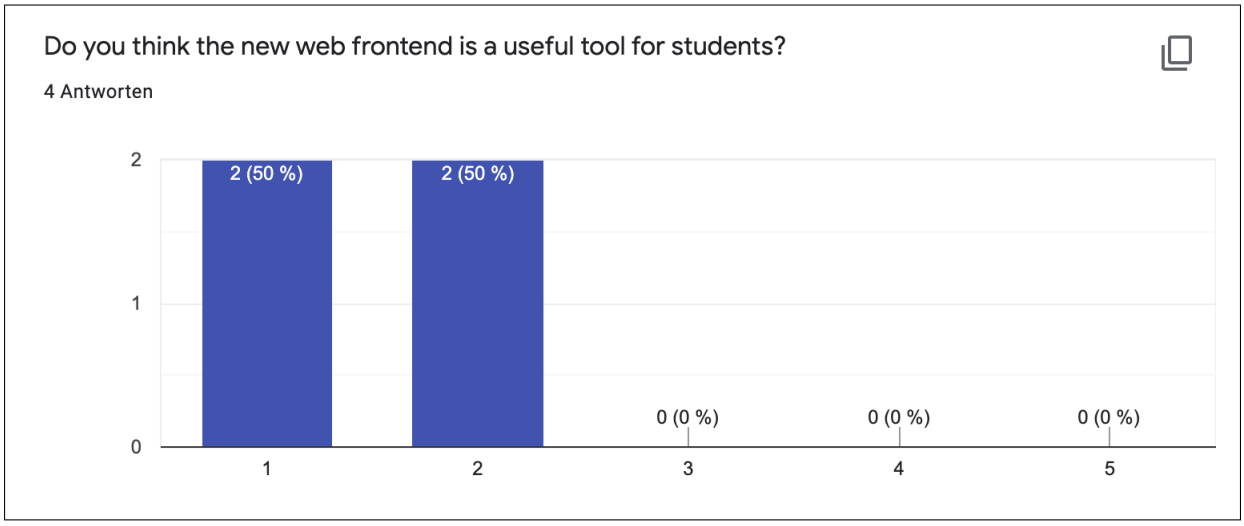
(1) Scale 1 = Yes, absolutely; 5 = Not at all



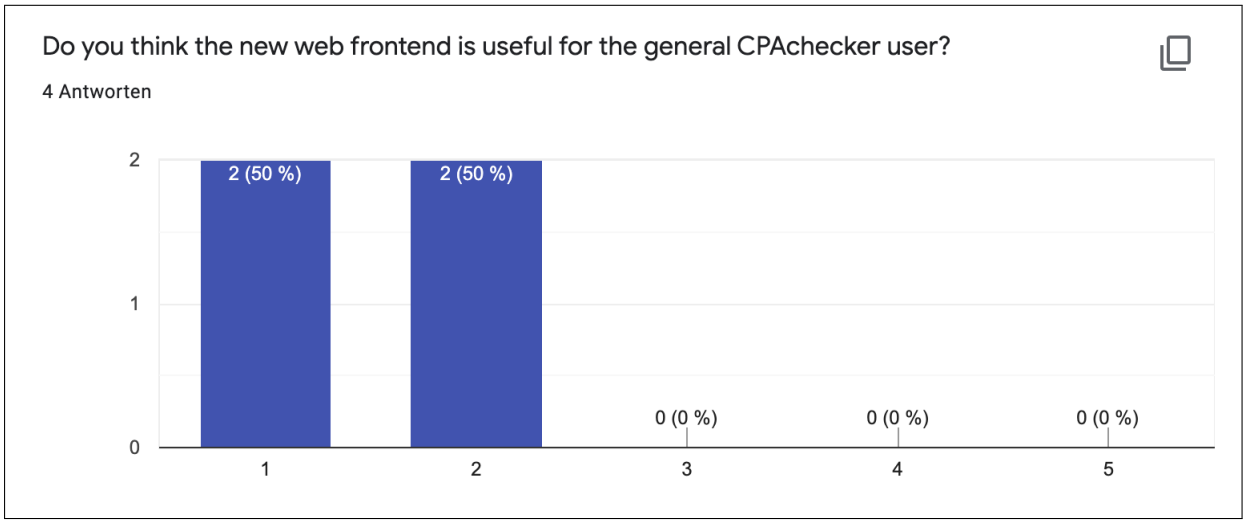
(2) Scale 1 = Absolutely; 5 = Still no clue



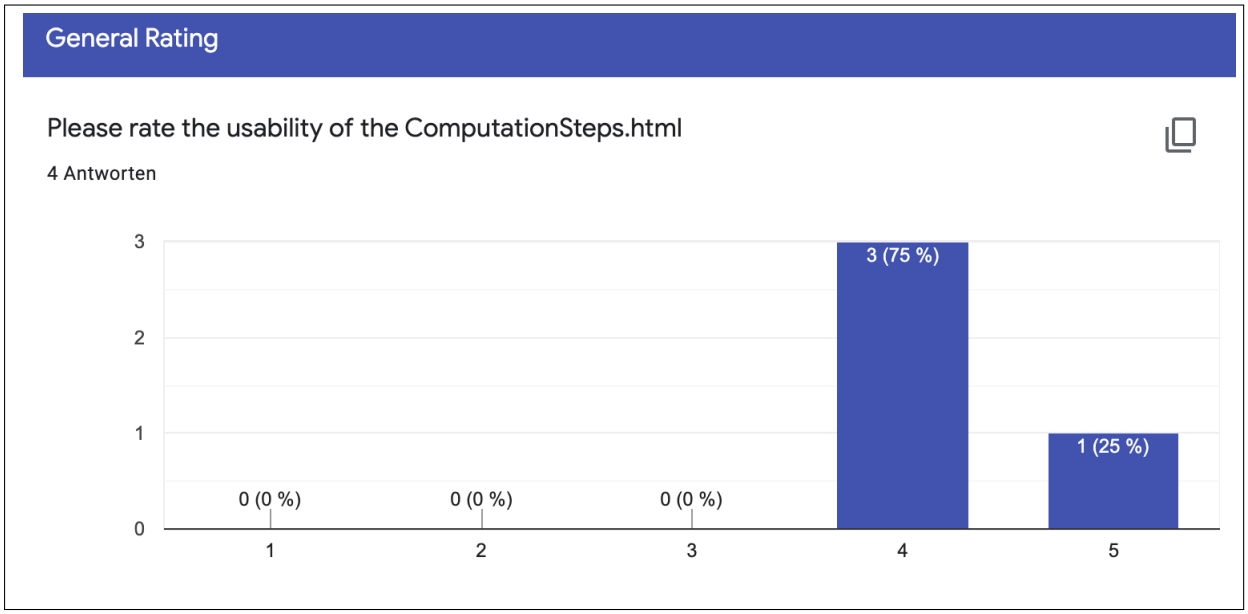
(3) Scale 1 = Yes, a lot; 5 = Not at all



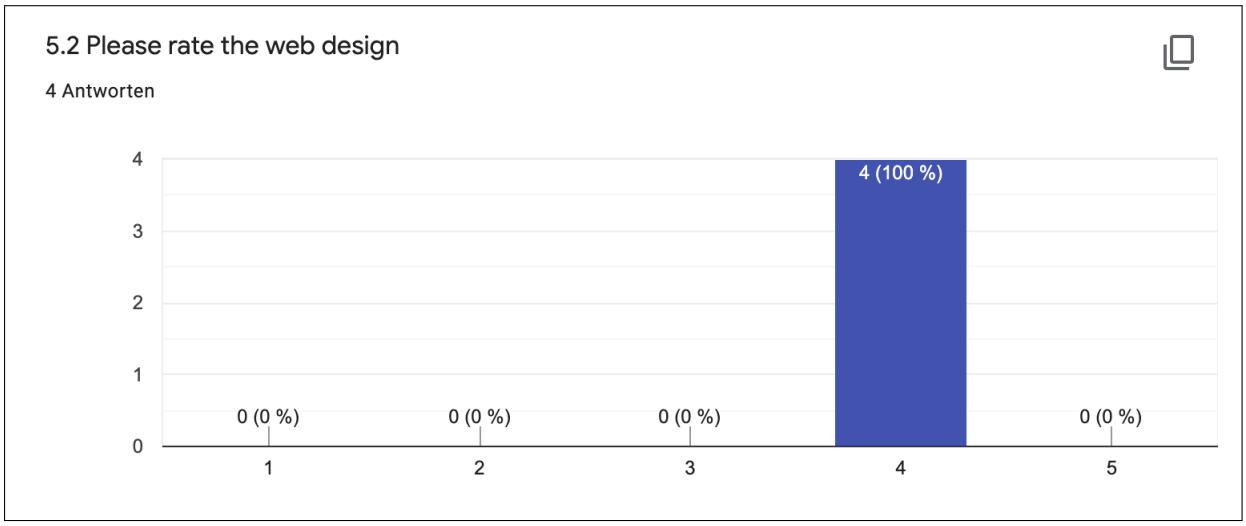
(1) Scale 1 = Yes; 5 = No



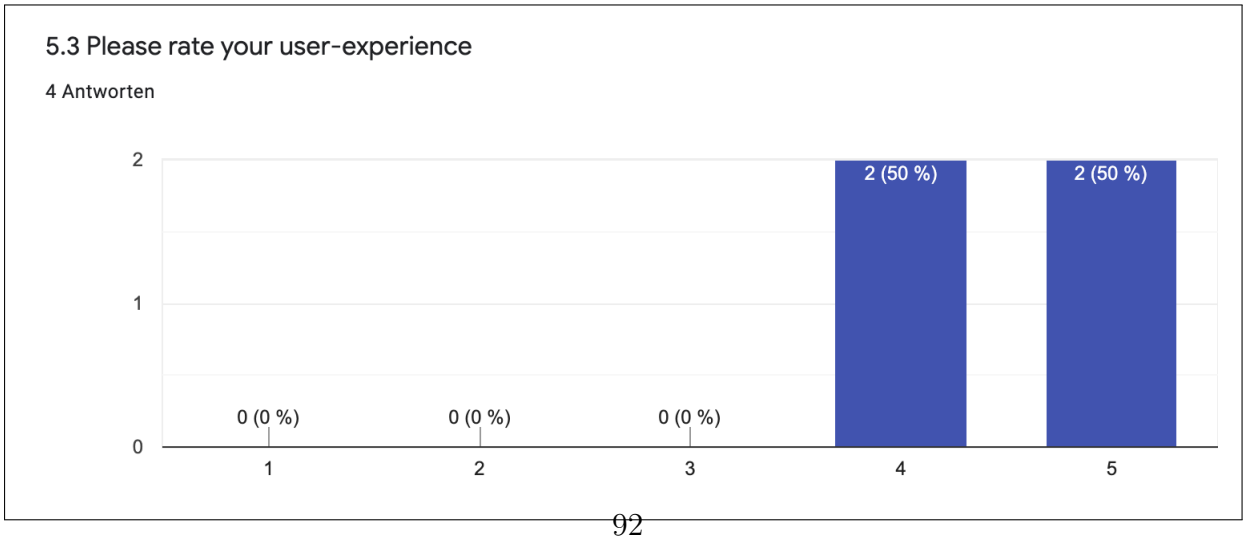
(2) Scale 1 = Yes; 5 = No



(1) Scale 1 = Completely unclear where to find and click; 5 = Everything was self-explaining



(2) Scale 1 = not attractive; 5 = attractive



(3) Scale 1 = It was a pain working with it; 5 = I enjoyed working with it