Bachelor thesis

Interactive Visualization of Verification Results from CPAchecker with D3

LMU Munich
Software and Computational Systems

Author: Deyan Ivanov

Professor:	Dirk Beyer
Supervisor:	Philipp Wendler
Munich:	10/17/2017

Abstract

CPAchecker is a tool for configurable software verification and is available for free under the Apache 2.0 License. It allows the verification of software that has been preprocessed with the C preprocessor. CPAchecker generates HTML report files depending on the verification outcome. The report includes graphical representations of the program flow and the reached abstract states, the source code of the program, the generated log entries and statistics as well as the used configuration options. In addition if an error is found by the analysis the generated counterexample report will include information about the program path leading to the error - the error path. This allows the user, by interacting with the generated report, to quickly and easily analyze the program and determine where exactly the error occurs and what its cause is. This document describes the latest implementation of the generated reports by CPAchecker which, as opposed to the previous solution, does not require the execution of an external script or the installation of an additional software. Additionally providing more interactive capabilities, such as pan and zoom functionalities for better graph readability and the option to display an abstract reachability graph containing only the error path, for the user. The solution uses the Dagre-D3 and D3 JavaScript libraries for graph creation and web workers for multithreading in JavaScript in order to be able to handle costly computational tasks on a background-running threads and thereby ensure performance and provide better user experience.

Contents

List of Figures

1 Introduction

CPAchecker[1] [2] is a configurable software-verification tool. It can be used to verify programs that have been preprocessed with the C preprocessor. CPAchecker generates HyperText Markup Language (HTML) reports that represent the verification outcome. A *Report.html* file is generated if there is no error found by the verification run and a *Counterexample.html* file is generated for each counterexample found by the system. Both report files include graphical representations of the control-flow automata (*CFA*), included in the CFA tab of the file and the reached abstract states (*ARG*), included in the ARG tab of the report. Additionally the generated report contains the source code of the analyzed program (*Source* tab), the generated log entries (*Log* tab) and statistics (*Statisctics* tab) as well as the used configuration options (*Configurations* tab). If a counterexample is found by CPAchecker, the report will also include a table-like representation of the error path, the path that leads to the error. To create the report, the user is required to execute an additional Python script that calls the Graphviz library in order to generate the graphical representations of the CFA and ARG. The goal of this thesis is to improve the generated verification results from CPAchecker as well as to reduce the known issues to a minimum. Improving the report generation can be achieved by removing the need to invoke additional software, meaning that the report will be automatically generated once the verification run terminates, thereby removing the dependencies to the Python script and the Graphviz library. Furthermore, this thesis aims to improve the generated report by providing more interaction and functional capabilities to the users, thereby assisting them to quickly and easily locate the source of an error or become a more detailed general overview of the analyzed program.

Before we continue with the details of the architecture and implementation of the report generation described in this thesis, a deeper understanding of the previous solution is required. Subsection 1.1 describes the available report-generation approach and discusses its advantages and disadvantages. Subsection 1.2 provides information about a software (named Klever) that is developed by a group of developers working on the CPAchecker project and aims to provide as much data for the error path as possible to the users if a counterexample is found. This thesis is further split as follows. Section 2 discusses the overall architectural approach consisting of the provided data from CPAchecker used for graphical representations of the CFA and ARG, presented in Subsection 2.1, and how web workers are used to achieve multithreading in JavaScript, presented in Subsection 2.2. Section 3 discusses the implementation approach in detail, presenting the logic for handling the report template files in Subsection 3.1, the used third-party libraries in Subsection 3.2, the graph-object generation details in Subsection 3.3, the use of web workers for result visualization in Subsection 3.4 and the used data binding in Subsection 3.5. Section 4 focuses on the generated report, its content and the available features. Subsection 4.1 provides detailed overview of the feedback presented to the users during graph creation. Subsection 4.2 presents the report-page layout and the available user-interaction capabilities in the generated report are contained in Subsection 4.3. This is followed by the performed evaluation and its outcome in Section 5 and ends with a conclusion and a discussion of possible future work and optimizations in Section 6.

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

1.1 Motivation

This section provides an overview of the currently existing report generation in CPAchecker and focuses on its advantages and disadvantages. The available solution is described in the bachelor thesis *Towards Understandable CPAchecker Counterexamples* by Magdalena Murr [6].

The current version of the report generation from CPAchecker provides some challenges to new users. Before the users are able to verify some C preprocessed software, aside from downloading CPAchecker[2], the users need to install a specific Python[3] version as well as download and install the graph-visualization software Graphviz[4]. Even though setting up the different software parts and the required environment variables is not particularly challenging for a software developer and is well documented, it can be off putting for a new user.

At the end of a verification run CPAchecker uses a HTML template file to generate the result report. The template contains the predefined report code as well as placeholders that are dynamically populated with data of the current verification run. The report file is named conveniently depending on the verification outcome - *Report* if the result is a success, meaning no error is found, or *Counterexample* otherwise. If more than one counterexample is found by CPAchecker during the verification run, a counterexample report will be generated for each one. In addition the verification run generates DOT[5] files. DOT is a plain-text graph-description language which leads to the generated files containing nodes and edges information for the CFA and ARG graph creation. Once the program run finishes, the users are prompted to execute a Python script that reads the generated report HTML file as well as the DOT files, calls the Graphviz library providing it the nodes and edges data contained in the DOT files and populates the resulting Scalable Vector Graphics (SVG) elements in the HTML file. After the Python script run finishes the users can view the generated report file by opening it in any web browser which will execute the JavaScript contained inside.

Figure 1 displays a generated counterexample report using the existing solution. The report is separated in six independent tabs which keeps the page clean and only provides one piece of relevant information to the user at a given time. The first is the *CFA* tab which contains a graphical representation of the control-flow automaton for each function contained in the verified program, displaying the main function per default and allowing the users to select and view a different one as well as providing them with the possibility to zoom in and out of the graph. The CFA graph contains special nodes called *combined* nodes. Those nodes contain linear sequences of "normal" edges (statement-edges, declaration-edges, and blank edges) and are represented as a rectangle with rounded edges in the graph. Additionally the CFA graph contains *function call* nodes that indicate the continuation of the program in a different function, which is displayed in a different CFA graph. The second tab, the *ARG* tab, includes a graphical representation of the abstract reachability tree and also provides the zoom functionality to the users. The other tabs are *Source*, containing a table-like representation of the source code, followed by *Log*, *Statistics* and *Configurations* which respectively provide the information their names suggest. If an error is found by the CPAchecker verification, the generated report page will include the source lines leading to the error in a table-like fashion contained on the left-hand side of the window, in addition to marking those program statements and states respectfully in the CFA and ARG graphs. The users are given the possibility to navigate through the error-path lines by either clicking

[2] `https://cpachecker.sosy-lab.org/download.php`
[3] `https://www.python.org/`
[4] `http://www.graphviz.org/Home.php`
[5] `http://www.graphviz.org/content/dot-language`

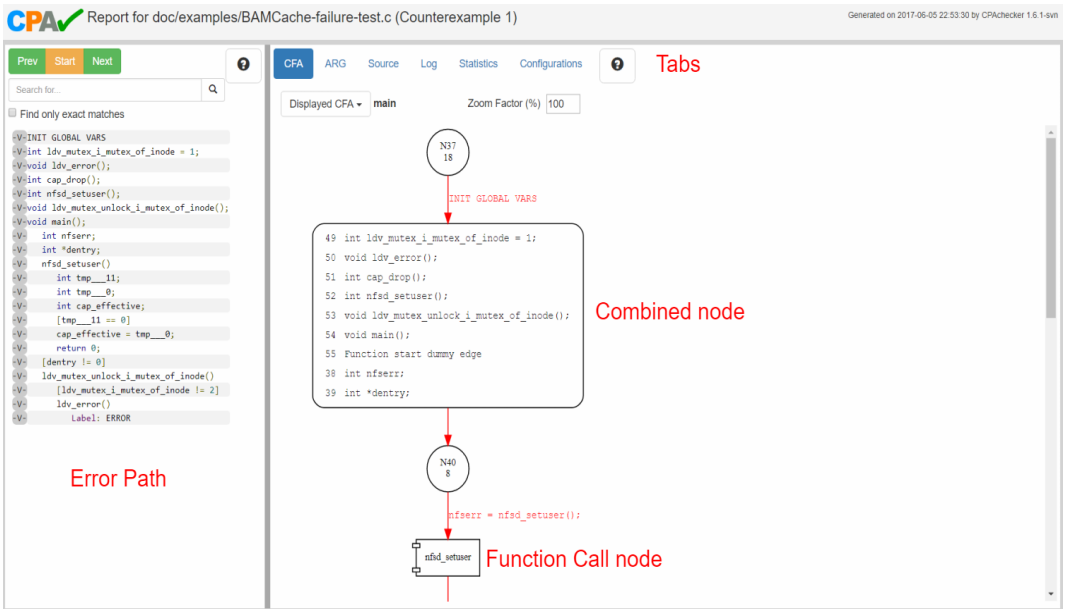

Figure 1: Counterexample Report - Existing Solution

on each of them or by using the *Prev*, *Next* and *Start* buttons, thereby assisting them to quickly and easily find where exactly the error occurs. Additionally the users can use the search functionality to allocate program variables, as well as their values and value changes, in the error-path table to further simplify the search of the error cause.

Due to the multiple steps that need to be executed before the final report is generated it requires time, especially for large and complicated programs, which result in large graphs that need to be created. Sometimes when the Python script is calling the Graphviz library to generate the SVG elements it does not terminate, so the user is forced to stop its execution. Even when the Python script terminates, opening the report in a browser may require a lot of time to render the SVG elements due to their size. Depending on the users' browser settings they will be prompted to make a decision if they want to stop the execution due to the page loading taking too long. Even after the SVG rendering finishes and the JavaScript execution comes to an end, there are still a few issues that the users encounter while interacting with the generated report.

The following subsection of the document provides an overview of the known issues present in the CPAchecker report. The list contains issues that were tracked by developers working on the CPAchecker project as well as additional problems that were found during the analysis of the report conducted as a part of this thesis.

1.1.1 Known Issues

The following list of known issues includes entries provided by developers working on the CPAchecker project that are available in the online documentation[6] as well as problems that were located during the examination of the available solution.

[6] `https://github.com/sosy-lab/cpachecker/blob/trunk/doc/BuildReport.md`

CFA error-path highlighting. The CFA tab displays the SVG element for one program function at a time providing the user with the possibility to change the displayed function by using the available dropdown menu. The SVG images are created during the Python script execution step of the report generation but the *error path* is highlighted dynamically using JavaScript. If the displayed function is changed, as a result of the user interacting with the report, sometimes the error path is not highlighted.

CFA combined nodes. There are two issues connected to the combined nodes. The available double-click event for "jumping" from a CFA edge to the corresponding line in the *Source* tab does not mark the correct source line for edges leaving the combined node. Additionally if the combined node contains too many labels it will lead to an error in the Graphviz library. To avoid the latter the labels for such nodes are replaced programmatically during the report generation with a general label stating *Long linear chain of edges between nodes*.

Error-path walkthrough. Selecting an element in the table-like error-path representation on the left-hand side of the report window will only update the currently active tab on the right.

Search in error path. The available search functionality in the error path does not always mark the error-path table rows, in which the searched variable is contained or in which its value changed, correctly.

ARG tab. When switching to the ARG tab it sometimes takes a couple of seconds to load the SVG image if the graph is too large. Additionally, if the graph is highly branched, the start node will be outside the presented viewport which leads to a white screen displayed to the users.

ARG error path highlighting. A similar issue to the CFA error-path highlighting problem listed above is observed for the ARG graph as well. Sometimes if the ARG tab is set as active and the graph is too large the error-path highlighting will not be included.

Zoom functionality. The zoom functionality available for the CFA and ARG graphs does not work as expected. It causes an "undefined" exception in the browser's developer console and does not affect the viewport of the graphs which leads to no change presented to the users.

Click events. The described "jump" events, in example from a CFA edge to a Source code line or from an ARG node to CFA node, do not always work as expected, in an essence that sometimes the marked element will not be contained in the viewport presented to the user or othertimes even ending in an error in the developer console and thereby displaying no change to the user.

The current report generation from CPAchecker with the additional Python script execution and the dependency on the Graphviz library as well as the known issues prompt to a different approach for the report generation that will provide an overall better user experience in regards of performance, usability and available features.

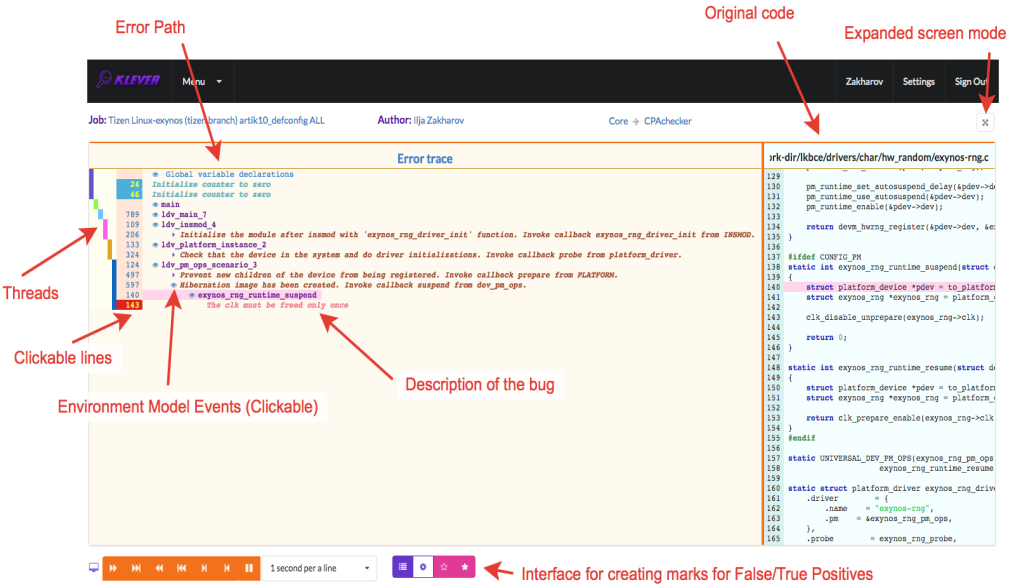

Figure 2: Klever

Source: Provided by Ilja Zakharov during a Web Conference at July 21st 2017

1.2 Related Work

This subsection describes an error-path focused report-generation approach that is developed by a group of software developers working on the CPAchecker project. The software is called *Klever* and its main goal is to keep information about the error trace as compact as possible in order to display it in a single screen and only further expand it on user interaction, in example via a click on an error-trace line. Klever is a web-based application developed with Django, which is a Model-View-Presenter framework written in Python, and is only available for a group of developers which is maintained through user management. This is a major difference in comparison to CPAchecker, Klever is not available to the public. The software is intended for arbitrary GNU C programs and provides the users with the possibility to schedule and run specific verification tasks on a predefined piece of software that is uploaded to Klever at a previous point in time.

Another major difference in comparison to the report generated by CPAchecker is that Klever does not provide any graphical aid to the user. According to one of the developers involved in the project there is no planning on including any graphical visualization. The verification-run outcome is presented in a textual matter allowing the user to expand or restrict the amount of displayed information by interacting with the view. The user interface makes use of several JavaScript libraries to display the data accordingly, in example *Bridge JS*[7], which exposes a service between one JavaScript context to another, *Treetable JS*[8], which is a *jQuery*[9] plugin used to display a tree like structure in HTML and *Notify JS*[10],

[7] `https://fxos-components.github.io/bridge/docs/out/index.html`
[8] `http://ludo.cubicphuse.nl/jquery-treetable/`
[9] `https://jquery.com/`
[10] `https://notifyjs.com/`

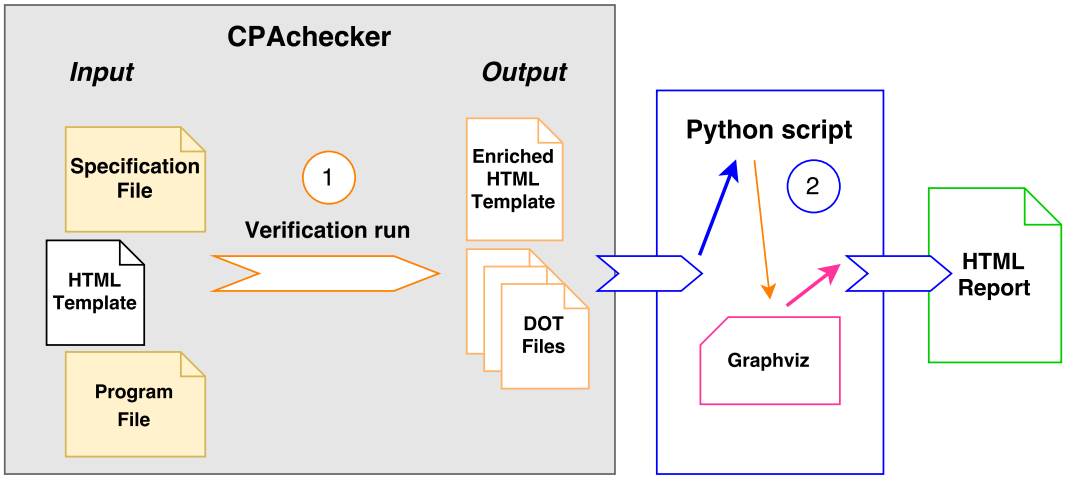

Figure 3: Previous Report-Generation Flow

which is another jQuery plugin used to provide notifications between different scripts.

Figure 2 displays the current evolution state of Klever. The left part of the screen shows the error-trace call stack that contains the code provided by a CPAchecker verification based on a C preprocessed file. The right part of the screen consists of the original C code or generated files. In order to compress the displayed data Klever uses different mechanisms. For example all generated or auxiliary code is hidden by comments of different colors. Brown for an environment model, blue for error-relevant functions and red for the error location. All this information is kept available and can be accessed by the users on demand by clicking on the eye and triangle icons in the user interface. Furthermore, if the users suspects the result to be a false positive or false negative they can mark it as such by using the toolbar on the bottom. All marked verification results can then be viewed by other users and respectively be acknowledged or discarded. Which on its own is argument enough to not make the software available to everyone.

In summary, Klever is a web-hosted application that can only be accessed and used by a group of developers and is thereby not available for public use. Additionally, Klever does not provide any graphical representation of the data, instead it displays the error path in a textual matter providing the users with the possibility to expand and compress the visible data on user interaction. This contradicts with the general idea behind the reports generated by CPAchecker, which are available to the public and provide graphical representations of the CFA and ARG data. Following this analogy it is not possible to exchange one software by the other or even come up with a unified solution that will satisfy both demands.

2 Architecture

This section focuses on the architecture of the new solution for the report generation by CPAchecker. As discussed in the motivation part of the thesis, Section 1.1, the current report generation presents some challenges to new users due to the fact that they need to install and run additional software to be able to view the result. Figure 3 displays the execution flow required for the report creation using the previous approach. The left-hand rectangle stands for CPAchecker where the HTML template is included in the source

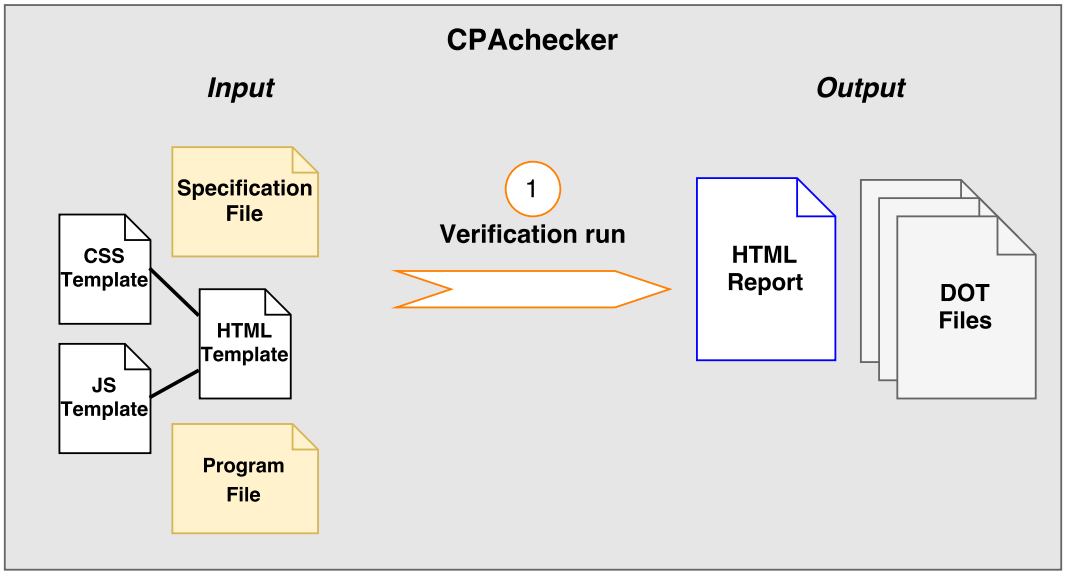

Figure 4: Target Report-Generation Flow

code. When the user triggers the verification run by passing a program file to be verified and a specification to be used for the verfication, marked with the number 1 in Figure 3, CPAchecker will enrich the HTML template file with additional data that is specific to the verification run and will create DOT files for each function contained inside the verified program. Those files contain the nodes and edges data required for the creation of the control-flow automaton and abstract reachability graphical representations. In order to generate the report the user is then required to execute a Python script that is included in the CPAchecker download. This step is marked with the number 2 in Figure 3. The Python script passes the previously generated DOT files to the Graphviz library which in turn generates the scalable vector graphics. The SVG elements are then embedded in the HTML file, thereby creating the final report which can be viewed and analyzed in a webbrowser.

In order to simplify and optimize the report generation by CPAchecker as much as possible a solution had to be found that will not require running the Python script, or any other additional software for that matter, after the verification run comes to an end. Since the Graphviz library is called by said Python script this means that ideally it should also be exchanged or atleast be called in a different way that does not require explicit user interaction. Optimally the report will be generated automatically by CPAchecker during the verification run. Furthermore, the goal was to keep as much from the existing architecture as possible in order to ensure a smooth and fast transition to the new solution. Since the report is essentially a HTML file that is viewed in a browser, the idea was to find a way to generate the CFA and ARG graphs with JavaScript as a part of the CPAchecker verification run. Figure 4 displays the target execution flow and architecture of the solution described in this thesis. As we can see Figure 4 consists of only one rectangle that stands for the CPAchecker verification process. The difference, in comparison to the left-hand rectangle in Figure 3, is that the new architecture uses multiple tempate files, HTML, CSS and JS, thereby honoring the separation of concerns principle in software development. When the user triggers the verification run, marked with the number one in Figure 4, CPAchecker will create the report HTML by combining the three template files into one file that can

be viewed in a browser without any further user interaction, thereby omitting the need to run the Python script and the Graphviz dependency. Additionally, the DOT files are still generated because some of the experienced users that are familiar with the previous solution might still want to analyze them. Nevertheless, the DOT files are not required for the CFA and ARG graph creation. In order to enable this approach the data required for the graph creation needs to be provided to the report by CPAchecker at the end of a verification run. Additionally, costly computational functions that are required for the graph-objects creation in the report file need to be moved to background-running threads in order to ensure better performance. Further details on how this can be achieved are discussed in the following subsections.

Subsection 2.1 contains the information about the JSON data provided by CPAchecker, focusing on its structure. Subsection 2.2 discusses the general use of *web workers* to achieve multithreading in JavaScript and focuses, furthermore, on the use of web workers in the CPAchecker report generation.

2.1 Data from CPAchecker

Removing the Python script and the dependency to the Graphviz library means that the generated DOT files, during the verification run, will not be used anymore. Since those files contain the nodes and edges data for the CFA and ARG graphs generation, a good way to provide this information directly to the HTML file needed to be found, more specifically in a way that said data can be used by the JavaScript code. The go-to standard in this case is the JavaScript Object Notation, JSON[11] for short. In order to achieve this at the end of the verification run, CPAchecker needs to provide the nodes and edges JSON data in the generated report file so it can be further used by the JavaScript code to generate the required graphs.

The last step in the CPAchecker verification run is the report generation, unless a specific parameter is passed by the user which will disable it. During this stage the provided HTML template is read line by line and enriched with the additional nodes and edges JSON data by following a set of predefined rules. Listing 1 displays the current structure of the data used to generate the control-flow automaton graphs for an example file called *BAMCache-failure-test.c*. The actual content of the file will not be discussed since it is not important for the structure of the generated JSON object. The CFA JSON object, as any JavaScript object, consists of key-value pairs. The usage of this data in order to generate the CPAchecker report is discussed in detail in the implementation part of this document, disclosed in Section 3.4. The line numbers in Listing 1 mark the keys contained in the CFA JSON object. The value behind the first key, *"functionNames"*, is a JavaScript *array*[12] containing the names of all functions included in the file BAMCache-failure-test. The value behind the key in line 2, *"functionCallEdges"*, is another object that carries information about functions called from within other functions and is used to specifically mark those locations inside a CFA graph. The next two keys, *"combinedNodes"* and *"combinedNodesLabels"* belong together. Similar to the key in line 2, the values are again objects that carry information about a specific node to be created in the graph. Behind the key *"mergedNodes"* is an array that carries the information about nodes that must not be displayed separately. The last three keys, *"errorPath"*, *"nodes"* and *"edges"*, hold arrays of objects. The error-path key is optional in a sense that it is not always available. This is the case if the verfication run can not find a counterexample for the verified program.

[11] `http://wiki.selfhtml.org/wiki/JSON`

[12] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array`

Listing 1: CFA JSON Structure


```
{
1. "functionNames":["main", ...],
2. "functionCallEdges":{"32":[100004,33], ...},
3. "combinedNodes":{"49":[50,51,52], ...},
4. "combinedNodesLabels":{"49":"49 int tmp___11;\n50 int tmp___0;\n51 int
    cap_effective;\n52 int tmp__6;", ...},
5. "mergedNodes":[50,51,52, ...],
6. "errorPath":[
{"val":"","file":"BAMCache-failure-test.c","line":3,"source":50,"target":51,"desc":"void
    ldv_error();"},
...],
7. "nodes":[
{"func":"ldv_error","loop":false,"index":1,"rpid":2,"type":"entry"},
...],
8. "edges":[
{"file":"BAMCache-failure-test.c","line":24,"source":19,"type":"DeclarationEdge","stmt":"int
    tmp___0;","target":20},
...]
}
```


The error-path objects provide information about the source-code lines that are part of the counterexample, marked with the key *"line"*, the *"source"* and *"target"* of the corresponding CFA graph edge and the executed program statement *"desc"*. The objects inside the *"nodes"* array hold information, as the name suggests, about each node, in example the function, *"func"*, in which the node is included, its *"index"* and its *"type"*. Similarly the *"edges"* array holds data about the graph edges, in example the *"type"*, *"source"* and *"target"* of the edge and the corresponding source-code *"line"* number.

Listing 2 displays the abstract reachability graph JSON object which, compared to the CFA JSON object, has a simpler structure. It only contains two keys, *"nodes"* and *"edges"* that hold arrays of objects providing the respective nodes and edges information for the ARG graph creation. Each node object carries data about the function in which the node is present, marked with the *"func"* key, the node's *"index"*, its *"label"* and its *"type"*. Each edge object carries information about the *"file"* in which the edge is contained, its *"source"* and *"target"*, the corresponding source-code *"line"* number as well as the edge's *"label"* and *"type"*. Even though the keys have the same denotation as the two keys contained in the CFA JSON structure, the data included in the object arrays is not the same. One additional important thing to note is that the *"errorPath"* data, contained in the CFA JSON object that is displayed in Listing 1, is also used for the ARG graph creation. Since the error path data does not deviate between CFA and ARG there is no need to duplicate it and include it in the ARG JSON object as well. Details on how the data from the CFA and ARG JSON objects is used to create the CFA and ARG graphs in the CPAchecker generated report are discussed in the implementation section of the thesis, see Subsection 3.4.

The next subsection of the thesis discusses *web workers* as means to execute JavaScript in multiple threads, focusing on their general use as well as their advantages and disadvantages.

Listing 2: ARG JSON Structure

```
{
1. "nodes":[
{"func":"main","index":0,"label":"0 @ N37\nmain entry\nAbstractionState: ABS0:
    true\\","type":"highlighted"},
...],
2. "edges":[
{"file":"BAMCache-failure-test.c","line":"51","source":93,"label":"Line
    51\n[!(dentry != 0)]","type":"AssumeEdge","target":96},
...]
}
```

2.2 Dividing the Work

JavaScript, although widely used and a must have part of any and all web-based applications, does present some challenges to the developers. One drawback is that the code is sequentially executed. This often results in the script taking too long to finish, which leads to the well known message asking the user to stop the page loading or to keep waiting for it to load. The previous report-generation implementation in CPAchecker suffered from this issue for particularly large and complex programs. One option to tackle this problem is asynchronous code processing, which basically simulates multithreading but is not very effective for computationally intensive code, for example graph creation and rendering. Another option is actual multithreading. This means storing and performing some computationally expensive operations on a thread running in the background, thus allowing the main thread, where the main script is being executed, to take care of the rest without causing the webbrowser to freeze. This can be achieved through the use of *web workers*. The next part of the document discusses the general use of web workers and how they are integrated in the CPAchecker report generation.

2.2.1 Web Workers

This chapter represents a high level overview of the general usage and available features of web workers. The contained information is taken and adjusted from the online documentation of the web workers API[13] from the mozilla developer network (MDN) and is explained using a self-written example.

Web workers allow web content to run scripts in background threads where a task can be performed without interfering with the user interface. The communication between a created worker and the main JavaScript is maintained by posting messages to an event handler specified at each end. Once the worker object is created using the constructor, in example *new Worker(fileName)*, it will run the JavaScript code contained in the provided file. One important thing to note here is that all workers run in another global context, which is different from the current *window*. Some webbrowsers, in example Google Chrome and Chromium, do not support the direct construction of a web worker using a separate file if the file is stored **locally**. In such case a workaround can be applied. It consists of defining the code that is to be run by the worker in a *self-invoking function* instead of a separate file and then use this function to create an in-memory file, a *Blob*, and pass it to the constructor. The web workers API differentiates between two types of workers, a

[13] https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API

dedicated worker and a *shared worker*. The difference between the two is that a dedicated worker can only be accessed by the script that spawned it, as opposed to a shared worker that can be accessed by any script. Furthermore, any worker can spawn additional workers, as long as those are hosted within the same origin as the parent page, thereby honoring the cross-origin resource sharing (CORS)[14] policy. A web worker can be used to run any code the developer wishes to, with some exceptions. For example, workers do not have direct access to the *document object model* (DOM)[15] or to some default methods and properties of the *window*[16] object. Some of the large number of items available to web workers include *WebSockets* and data-storage mechanisms like *IndexedDB*. WebSocket[17] is a communication protocol that provides a full-duplex communication channel over a single TCP connection that can be used in example for development of user chat features in web applications. The IndexedDB API[18] provides the possibility to store significant amount of structured data, including files and blobs on the client side. Functions available to the *WorkerGlobalScope* include *setInterval()* and *clearInterval()*, *setTimeout()* and *clearTimeout()*, *importScripts()* and *postMessage()*. The *setInterval* function is used to repeatedly call a function or execute a code snippet with a fixed time delay between each call. Using *clearInterval* the repeated execution from *setInterval* can be stopped. The method *setTimeout* sets a timer which executes a function or specified piece of code once after the timer expires. The set timer can be cancelled by calling the function *clearTimeout*. APIs available in workers include *Cache* which provides storage mechanisms for request and response object pairs on the client side, *Broadcast Channel API* which allows communication between browsing contexts, in example windows, tabs and frames with the same origin policy. Additionally, Web Workers can make use of *Custom Events* which provides the users with the ability to create, trigger and listen to events defined by themselves, *FileReader* which allows asynchronous read of blobs and files honoring the same origin policy and many more. For the full list of available functions and APIs to web workers please visit the *Functions and classes available to Web Workers*[19] part of the MDN web worker API documentation. Two of the available functions mentioned previously, the *importScripts()* and *postMessage()*, require further examination. The first, as its name suggests, is used to import scripts in the web worker. This includes self-written JavaScripts and third-party libraries where in case of own content the parameter provided to the function call can be a filename or a self-invoking function. In case of a third-party library the parameter passed to the importScripts function must be the library's URL. The *postMessage()* function is the bread and butter for web worker usage, it handles the communication between the main script and the worker script. The parameter passed to the function can be a simple string or a JSON string. Following is a basic, self-written, example on how the two scripts, running on separate threads, communicate with each other. Please note that the example displays the use of dedicated web workers and not shared web workers.

There are a couple of important things to note in the example Listing 3. The code contained between the lines 1 and 6 must be included in a separate file, in this case called *webworker.js*. It adds an event listener to the web worker itself, note the keyword *self* that listens for the occurance of a *message* event. The message event carries the message data in a property, conveniently called *data*. An important thing in this context is that the actual

[14] `https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS`
[15] `https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model`
[16] `https://developer.mozilla.org/de/docs/Web/API/Window`
[17] `https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API`
[18] `https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API`
[19] `https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Functions_and_classes_available_to_workers`

Listing 3: Web Worker communication example

```
1. // in a webworker.js file
2. self.addEventListener('message', function(m) {
3.    if (m.data === "Hello") {
4.       self.postMessage("Hello to you too!");
5.    }
6. })
7.
8. // in the main script
9. var someWorker = new Worker("webworker.js")
10.   helloWorker.postMessage("Hello");
11.   helloWorker.addEventListener('message', function(message) {
12.      if (message.data === "Hello to you too!") {
13.         alert("Worker said: " + message.data);
14.      }
15.   })
16.
17.   helloWorker.addEventListener('error', function(e) {
18.      alert("Worker error at line " + e.lineno + " with message " + e.message);
19.   })
```

message is not shared, it is copied between the two scripts. In line 3 we examine the data property and respond appropriately back to the main script, note again the *self* keyword in line 4. The reason for the use of *self* is that the worker itself is effectively the global scope instead of the usual window global scope. This is also the reason why a web worker does not have access to the window object. Line 9 spawns the dedicated worker, called *helloWorker*, with the code contained in the webworker.js file by passing the file name to the constructor. Now this worker can only be accessed by our main script. Once we have created the worker, we send the message containing the data *"Hello"* to it, as shown in line 10, by invoking the *postMessage()* function. The web worker responds accordingly as defined in lines 2 through 6 and the main script receives the message object in line 11, were the actual message, *"Hello to you too!"*, is again contained in the data attribute. In case an error occurs during the code processing inside the worker, an *error event* will be fired. In order to react, the main script must define an additional event listener for the worker object that will listen for the error event, see lines 17 through 19. In this case our example Listing 3 will display the error to the user.

The setup in the example expects both files to be contained in the same directory. If this is not the case, the parameter in the worker constructor can be adjusted to include the path to the file or the file URI. When using a file URI, it is important that it obeys the same-origin policy as the main script. The example above does not include one additional important topic and that is the termination of a worker. This can be achieved in the main script by calling the worker's terminate method, in example *myWorker.terminate()*. On the other hand a worker can close itself by calling the *close()* method. This points to the second big advantage of web workers, the first one is that they run in a separate thread. A web worker can continue to run even if the main script, the script that spawned the worker, has been terminated. If it is a dedicated worker and not a shared worker, no other script will be able to communicate with it. Nevertheless, this is still a big advantage, for example if the web worker needs to write to a database or create a file using the web-worker available APIs,

because it will be able to finish the task, even if the main script is terminated. Furthermore, dedicated web workers are widely supported by the most well known and used webbrowsers, in example FireFox, Google Chrome, Opera, Edge and others. Shared web workers, on the other hand, are only available in FireFox and Google Chrome. The full list of supported browsers and versions is available in the mozilla developer network online documentation[20]. One additional important thing to consider, when working with web workers, is *thread safety* due to the fact that the worker interface spawns operation-system level threads. Since web workers have no access to the DOM, the only thing a developer has to ensure is a flawless communication between the threads.

Now that the idea behind web workers and their usage has been presented, we will analyze, in the next part of the document, how web workers are used for the generation of the CPAchecker report.

2.2.2 Web Workers in CPAchecker Report

The main script in the CPAchecker report spawns two web workers, called *cfaWorker* and *argWorker*. They take care of the control-flow automaton and the abstract reachability tree graph-objects creation respectively, thereby taking away the costly graph-object creation tasks away from the main script. Since the web workers do not have access to the DOM, the graph rendering is executed by the main script. If an ARG is not available because the CPAchecker analysis was not based on ARG states, there is no ARG data provided and the ARG graph can not be build. In such cases the argWorker is also not created.

As mentioned in the previous subsection, the communication between the workers and the main script is a crucial thing to consider when using multiple threads in JavaScript. In order to ensure this, it is advisable to carefully plan and create a communication protocol for the threads. Figure 5 visualizes the communication protocol created to fit the communication needs between the main script and the two workers used during the generation of the CPAchecker report. Please note that this is an overview of the transfered messages and their timing between the threads. It does not provide any information about the implemented logic handling and reacting to those messages. This is contained in the implementation section of the thesis, Subsection 3.4.

Figure 5 displays the ARG worker on the left, marked in red, the main script in the middle, marked in green and the CFA worker on the right, marked with blue. The text contained in each line in its respective block provides a brief explanation of the executed action. Note the pink-ish marked text, *User Interaction*, displayed in about the middle of each block. This marks the spot from which all actions contained below are only executed if a specific trigger occurs. Meaning, if the user interacts with the user interface in a special way. The arrows connecting the blocks represent the source and target of the message. Additionally, the color of the arrows denotes the source of the message. Dottet arrow lines stand for optional message transfer. As shown in Figure 5 this marks all user interaction actions, which is no surprise because an user can choose to interact with the report or not. As explained above, sometimes there is no ARG data available. In such cases the ARG Worker will not be created, thus the whole left part of the figure will be omitted. In cases where ARG data is available but no error was found by the analysis, meaning there is no counterexample, the main script will not provide the error path data to the ARG worker, because it does not exist. This is the reason for the arrow marked with the number 2 being dotted, which again stands for optional. The numbers above the lines represent the sequence in which the messages are transfered from sender to recipient. Please note that

[20] `https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API`

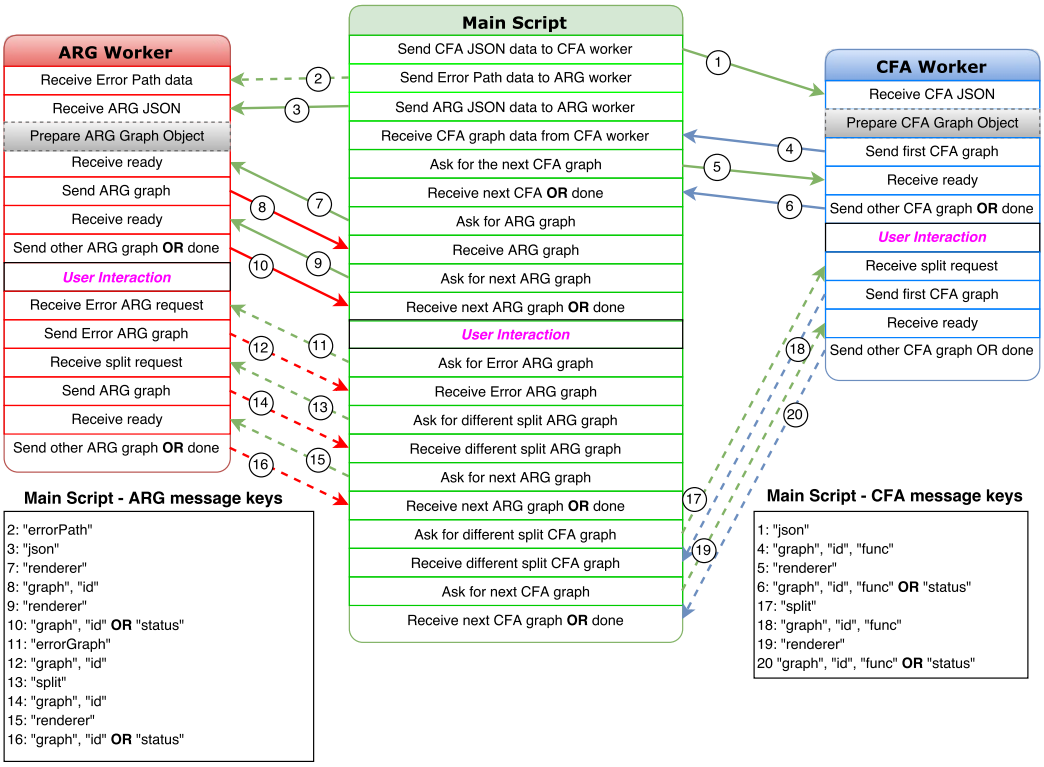

Figure 5: Main Script - Worker Communication Protocol

the numbers below the *User Interaction* mark are kept sequential only to provide hooks for better explanation of the executed actions. The boxes below the blocks, on the left and on the right sides, contain the message keys, matching them to the arrows by using the numbers in each line. The protocol transfers the messages in JSON objects, hence the object keys. The values are not included here because they are not important for the communication and will, therefore, be explained in detail in the implementation section of the document, Subsection 3.4.

Lets assume that the CPAchecker verification has ended and a counterexample has been found for an analysis that contains ARG data. When the report is opened by the user, in a browser of his choice, the main script will be triggered and will spawn the two workers which will take care of the CFA and ARG graph objects creation. Immediatelly after the main script will send the first message, with the key *"json"* to the CFA Worker, please note the arrow marked with the number 1. This message is used by the CFA Worker to start preparing the CFA graph data. The graph-object creation task is marked in the figure 5 with a gray background line. Step 2 is providing the error-path data to the ARG Worker, message with key *"errorPath"*, and it is followed by providing the message with key *"json"* to the same worker in step 3. Please note that this message is not the same as the one send to the CFA worker in step 1. Those two messages are used by the worker to start preparing the ARG graph objects. This is marked with a gray-background line in the ARG worker block. In the next step, step 4, the CFA Worker provides the first CFA graph to the main script in a message containing the keys *"graph"*, *"id"* and *"func"*. Once the main script processes this information it will request one further CFA graph from the CFA worker with the message containing the key *"renderer"* in step 5. The CFA worker will respond with the same message as in step 4 or, if no more graphs are available, it will respond with the message key *"status"*, marked with the number 6. This basically means that steps 4 and 5 are executed as long as there are CFA graphs that need to be send to the main script for rendering, if this is no longer the case, step 6 will be executed. Immediatelly after receiving the message key *"status"* from the CFA worker the main script will send the massage marked with the number 7, containing the key *"renderer"* to the ARG worker. This will trigger the response with keys *"graph"* and *"id"* marked with the number 8. Once the main script is done processing the data it will send the message key *"renderer"* again to the ARG worker, number 9. If there are more ARG graphs available, the worker will send them using the message from step 8 which will again lead to the message from step 9. If this is not the case, the ARG worker will respond with the message key *"status"* in step 10. On a side note, the reason behind having multiple ARG graphs is a technical one, details are disclosed in the next section, see Subsection 3.4. At this point the CFA and ARG graph data has been provided to the main script and is being displayed to the users. The next message, marked with the number 11, is an user interaction requesting an ARG graph that only consists of the error-path data. When the message with the key *"errorGraph"* is send to the ARG worker, the response will contain the keys *"graph"* and *"id"*, similar to step 8, which is marked with the number 12 in Figure 5. Furthermore, the user can request a different ARG graph split ratio which will result in the main script sending the message key *"split"* to the ARG worker. Again the reason for the split ratio is technical and is explained in detail in Subsection 3.4. Sending this message to the ARG worker will trigger the recreation of the ARG graphs and a chain message exchange between the worker and the main script until all graph data is provided to the main thread, ending with the message key *"status"*. This is represented by the numbers 13 through 16 in Figure 5. The same split option is also available for CFA graphs. In this case the main script will send the message key *"split"* to the CFA worker causing similar effect as previously explained. This is represented by numbers 17 through 19

in Figure 5.

The communication between the main script and its workers follows the well known sender-recipient model. Even though the one explained in the previous lines seem tailored specifically for the needs of the CPAchecker report generation, it can be used for different implementations. This approach allows easy exchangeability of the message handling logic while keeping the same message keys. This means that *"how a message is processed"*, is not affected by the *"do I even react to this message"* and vice versa. The message handling logic for the reports generated by CPAchecker is explained in detail in Subsection 3.4 of the document.

3 Implementation

This section of the document focuses on the implementation details of the new solution for the reports generated by CPAchecker. Subsection 3.1 provides details on how the HTML, CSS and JavaScript templates contained in CPAchecker are used to generate the interactive report. Subsection 3.2 provides an overview of the third-party JavaScript libraries used in the implementation. Subsection 3.3 contains a detailed explanation on how the CFA and ARG graph objects are created using the third-party libraries. Subsection 3.4 focuses on the implementation details of the message exchange between the main script and the web workers introduced in Subsection 2.2.2. Subsection 3.5 explains the the *data binding* used to connect the view and the model in the new CPAchecker report and Subsection 3.6 provides an overview of one advantage achieved by the new implementational approach, the interchangeability of the used third-party libraries.

3.1 Report Templates

In the available solution for generating CPAchecker reports the HTML template also contains the CSS and JavaScript code, which makes it challenging to maintain and debug. This also contradicts with one of the important principles in software development, the separation of concerns. This approach did bring one major benefit. It provides the possibility to send the generated result to other users, in example via e-mail, since everything is contained in one file and in order to view it the users can directly open it in a browser. This advantage needed to be made available in the new implementation as well. The solution presented in this thesis follows a different approach regarding the used template file for report generation. The HTML template file no longer contains the CSS and JavaScript code. The three code modules are split in different template files as shown in Figure 4, thereby providing better debugging possibilities to the developers and honoring the separation of concerns principle. At the end of a CPAchecker verification run the HTML template file is read line by line, during which, if the current line contains a predefined string it will trigger a specific action from CPAchecker. Those lines are included as HTML comments in the template. Listing 4 displays the predefined strings contained in the HTML template that will trigger specific actions by CPAchecker, or more specifically the *ReportGenerator* Java class that contains the logic for the report generation, as its name suggests. It is important to note that this is not the only code contained in the HTML template, Listing 4 only displays the predefined strings that trigger the occurance of a distinct action. The activities executed by the ReportGenerator Java class are connected to inserting dynamic data, data that varies between different verification runs, into the HTML template. The <*head*> part of the document, displayed between the lines 1 and 5 in Listing 4, contains only three action causing strings. The first one is the *METATAGS* in line 2 and it writes the current CPAchecker version

Listing 4: HTML template action strings

```
1. <head>
2.   <!--METATAGS-->
3.   <!--REPORT_CSS-->
4.   <!--REPORT_JS-->
5. </head>
6. <body>
7.   <!-- REPORT_NAME -->
8.   <!-- GENERATED -->
9.   <!--SOURCE_CONTENT-->
10.  <!--LOG-->
11.  <!--STATISTICS-->
12.  <!--CONFIGURATION-->
13. </body>
```

Listing 5: JavaScript template action strings

```
1. var argJson={};//ARG_JSON_INPUT
2. var sourceFiles = []; //SOURCE_FILES
3. var cfaJson={};//CFA_JSON_INPUT
```

number as a HTML *meta*[21] tag in the generated report. The value contained in line 3, *REPORT_CSS*, causes the ReportGenerator class to read the defined CSS template, wrapped in a <*style*> tag, and insert it line by line in the HTML report. The string *REPORT_JS* displayed in line 4 in Listing 4, similarly to REPORT_CSS, triggers the inclusion of the JavaScript template, wrapped in a <*script*> tag, in the HTML report. The JavaScript template also contains some predefined values that cause the execution of specific actions. Before we continue with the analysis of the predefined strings that trigger diverse actions during the CPAchecker report generation contained in the lines 6 to 13 in Listing 4 we will focus on the values included in Listing 5. It is import to note that this is not the only code contained in the JavaScript template used for the CPAchecker report generation but it is the relevant one causing the insertion of dynamic data that varies for each verification run. Line 1 in Listing 5 that contains the value *ARG_JSON_INPUT* causes the ReportGenerator Java class to write the structured JSON data required for the abstract reachability graph creation, if it is available for the verification run, as shown in code Listing 2 prefixed with the variable declaration *var argJson*. The *SOURCE_FILES* value triggers the insertion of a JavaScript array containing the information about the *source files* on which the verification was executed, prefixed with the respective file path, in example *["doc/examples/bubble_sort_false-unreach-call.i"]* where the verified file is called *bubble_sort_false-unreach-call.i* and is available in the folder *doc/examples*. Line 3 in Listing 5 is handled similarly to line 1 of the same listing. It causes the Java class ReportGenerator to insert the structured data required for the control-flow automaton graph creation, as displayed in Listing 1, in to the JavaScript template. Now that the dynamic data insertion in the JavaScript template file is concluded we can continue with the analysis of Listing 4. Line 7, contained in the HTML body[22] tag, causes the insertion of the file names, on which the verification was performed, postfixed

[21] https://developer.mozilla.org/en-US/docs/Web/HTML/Element/meta
[22] https://developer.mozilla.org/en-US/docs/Web/HTML/Element/body

with the report name, either *Report* if there is no counterexample found by the verification run or *Counterexample* otherwise. The *GENERATED* string, contained in line 8 of Listing 4 inserts a timestamp representing the date and time at which the verification was performed, additionally including the current CPAchecker version number. The *SOURCE_CONTENT* value triggers the insertion of a table-like representation of the source code included in the file on which the verification is performed, additionally honoring indentations to provide better readability to the users. Line 10 in Listing 4 ensures that the generated *LOG* output for the verification run by CPAchecker is inserted in the report file. Line 11, containing the entry *STATISTICS*, triggers the insertion of the statistics data provided by CPAchecker. This includes, amongst other things, information about the compilation time required for each step that CPAchecker executes during a verification run and the used memory. Finally the line marked with number 12 in Listing 4 ensures that the *CONFIGURATION* options passed to CPAchecker at the start of the verification run are also made available in the generated report file.

The report file creation process follows the same logic as in the previous solution in regards that it looks for predefined strings that trigger the execution of a specific action. All comment lines and the correspondent actions included in Listing 4, except the ones in line 3 (REPORT_CSS) and line 4 (REPORT_JS), are kept from the previous implementation. On the other hand the lines *CFAFUNCTIONGRAPHS* and *ARGGRAPHS* from the old solution are not required anymore. Those strings were the trigger for the insertion of the SVG elements generated by Graphviz during the Python script execution. Additionally, the previous solution was receiving the data required for the creation of the CFA graphs in the JavaScript part of the HTML template. The main difference here in comparison to the implementation presented in this thesis is that essentially the same graph data as shown in Listing 1 was provided in autonomous JavaScript arrays and objects. Furthermore, the data required for the ARG graph creation as shown in Listing 2 was not provided to the generated report in the old implementation.

To summarize the approach in the new implementation, the code parts taking care of what the report contains (the HTML), how it looks like (the CSS) and how it behaves (the JavaScript) are separated in different files, honoring the separation of concerns principle but still ensuring that the generated report is only one file that can easily be transfered between users. During this process CPAchecker, or more specifically the ReportGenerator Java class, exports the data required for the report generation in a structured way.

3.2 Third-Party Libraries

This subsection provides an overview of the third-party libraries used for the CPAchecker report generation and their general use.

3.2.1 D3 JS

D3[23] stands for Data-Driven Documents and is a JavaScript library that provides convenient functions for document manipulation based on data. It helps its users to bring data to live using SVG, Canvas and HTML by combining powerful visualization and interaction techniques with a data-driven approach for DOM manipulation. The core components of the D3 library include *Selections*, *Transition*, *Collection manipulation functions*, *String and CSV Formatting functions* and others. A selection is an array of elements obtained from the current document which is retrieved by using CSS3 selectors. Elements can be

[23] `https://D3js.org/`

selected in various ways, in example by a HTML tag ("p"), CSS class (".myClass"), attribute ("[width=30]") or unique identifier ("#myId"). Once the elements are selected D3 provides the possibility to apply operators to them which enable the users to get or set styles, attributes, properties, HTML and text content. A transition is a type of selection derived by using the *transition* operator. The difference between a selection and a transition is that with a transition the operators are applied over time with a specified delay and duration as opposed to a selection where the operators are applied instantaneously. D3 provides the possibility to apply different delay and duration per element which will result in different transitions based on data. As mentioned before, D3 selections retrieve arrays of elements which means that the user often needs to do array manipulation. The library provides a range of helper methods that include *mutator methods* for adding, removing and sorting array elements, *accessor methods* for concatenating, extracting a section or finding a specific element in the array as well as *iteration methods* for filtering and traversal of the elements. The *formating methods* mentioned previouosly include helper functions for easy number formating as well as reading, parsing and manipulating comma-separated values (CSV). Additionally, the D3 JS library provides a high amount of graph layouting and drawing capabilities that can be used when working with scalable vector graphs or HTML canvas. The available layout options include, amongst others, *Cluster* which produces dendograms - node-link diagrams that place leaf nodes of the tree at the same depth, *Force* which provides the possibility to build graphs for which the contained nodes attract or repel each other based on data and *Histogram* which can be used to show the distribution of data by grouping its discrete points into bins. For drawing the library provides, amongst other things, *lines*, *shapes* and *symbols* drawing methods which can be used to customize the look of the created graphs. Furthermore, D3 JS provides convenient *behavior* methods that can be attached to elements in order to enable the user *drag* and *zoom* interaction features. For further information please visit the D3 API[24] online documentation.

3.2.2 Dagre D3

Dagre-D3[25] is a D3-based renderer for *Dagre*[26] and Dagre is a JavaScript library that focuses only on the layouting of directed graphs while prioritising *speed* in order to draw medium sized graphs quickly. This means that once the layout information is obtained by *Dagre* a renderer is required in order to create the graph which is where Dagre-D3 comes in. The skeleton of Dagre is based on a four-step algorithm [4] for drawing directed graphs. The first step of the algorithm assigns optimal ranks to the graph nodes. The second one sets the vertex order within the ranks determined in step one. The third step calculates the optimal coordinates for the graph nodes and the final step creates splines to draw the graph edges. In addition to the basic skeleton derived from the paper, Dagre specifically uses the described method to internally produce an acyclic graph from the input graph and the *network simplex algorithm* for node ranking. The network simplex algorithm assigns initial ranks to each node and iteratively improves the ranking to reduce the length of the edges. A high level overview of the algorithm consists of three steps. First an initial rank is assigned to each node which is achieved with the longest path algorithm[27] in Dagre, thereby assigning ranks to the lowest position possible. In the second step a feasible tight tree is constructed, so that for all edges in the tree the actual edge length matches the minimum edge length. In the third step the network simplex algorithm iteratively checks for edges with *negative*

[24] https://github.com/D3/D3-3.x-api-reference/blob/master/API-Reference.md
[25] https://github.com/cpettitt/Dagre-D3
[26] https://github.com/cpettitt/Dagre/wiki
[27] http://www.geeksforgeeks.org/find-longest-path-directed-acyclic-graph/

cut value which indicate that the edge can be replaced with a new edge in order to produce a more compact graph. To determine the node order Dagre uses heuristics to minimize edge crossings in the graph [1], [5]. Finally Dagre uses a three-step algorithm to assign coordinates for the nodes [3]. The first step, *vertical alignment*, aims to align each vertex with either its median upper or its median lower neighbor. Resulting conflicts are resolved by Dagre in a rightmost fashion. In the second step, called *horizontal compaction*, the aligned vertices (nodes) are constrained to obtain the same horizontal coordinate. Additionally, all vertices are placed as close as possible to the next vertex while taking the predefined minimal separation value into consideration. The first two steps are carried out four times and in the last step the, thereby, obtained four assignments are combined to balance their biases. During node rank assignment and the vertex ordering Dagre adds artificial nodes to the graph, *dummy* nodes. Those dummy nodes are used to determine points for the drawn splines before they are removed from the final graph.

3.2.3 AngularJS

AngularJS[28] is a JavaScript framework that follows a MVW pattern which stands for *Model-View-Whatever* as in "whatever works for you". This indicates that the framework provides the users with the capability to choose from the pattern options *Model-View-Controller* (MVC), *Model-View-Presenter* (MVP) and *Model-View-ViewModel* (MVVM) the one that fits their needs best. An application build with AngularJS usually consists of a HTML file which represents the view and JavaScript code containing one or multiple *Controllers* grouped in a *Module* that take care of the data management logic. The main concept behind the AngularJS framework is the *two-way data binding* which represents a bidirectional connection between the view and the model. This means that whenever the model is updated the changes will be reflected in the view and vice versa. The framework thereby takes care of DOM manipulation tasks, like updating the DOM elements, registering callbacks and listening for model changes, automatically. This eliminates those tasks from the list of things the users need to worry about.

AngularJS realises two-way data binding through *Directives* and *Expressions*. An example of an expression embedded in the view can look like this <*p*>{{ *counter* }}</*p*>. It represent a HTML paragraph tag (*p*) containing the string *counter* inside a double set of curly braces. The curly braces tell AngularJS that "counter" is not just a string but a model variable which causes the framework to automatically set a *watcher* on the variable. If the value of the variable "counter" changes in the model the watcher will take care of updating the view to present it to the user. Directives provide reusable code for DOM manipulation. For example the *ngIf* directive can be used to display or omit a portion of the DOM based on an expression, the *ngRepeat* creates a template for each item from a collection and the *ngClick* directive executes custom behavior on a click event. The online API[29] documentation contains, amongst other things, a list of all available directives as well as information on how to create custom directives.

3.2.4 jQuery

The JavaScript library jQuery[30] provides convenient methods for HTML document traversal and manipulation, event handling, animations and the execution of asynchronous

[28] `https://angularjs.org/`
[29] `https://docs.angularjs.org/api`
[30] `https://jquery.com/`

tasks. The jQuery API can easily be spotted due to its unique namespace, the $ sign. It provides comfortable DOM element selection functions, in example by tag *$("div")*, by CSS class *$(".myClass")*, by attribute *$("[widht=30]")* or by unique identifier *$("#myId")*. The used selection criteria can also be accumulated or disjoint. Once an HTML element is selected, jQuery provides a high number of methods for the retrieval or manipulation of its CSS classes and attributes. Those methods include adding - *$("mySelection).addClass('myClass')"* or removing - *$("mySelection).removeClass('myClass')* a CSS class. Additional functions enable the retrieval of the value of a specific attribute - *$("mySelection").attr("height")* or updating the attributes value - *$("mySelection").attr("height", 100)*. Furthermore, jQuery provides convenient methods for adding event listeners to elements, in example *$("mySelection").click(function() {...})* will add an event handler function to the selection that will listen for the occurance of a click event. Similar code syntax can be used to programmatically trigger an event, in example *$("mySelection").click()* will execute a mouse click on the element matching the selector "mySelector". For further information please visit the online documentation of the jQuery API[31].

3.2.5 Bootstrap

Bootstrap[32] is a HTML, CSS and JS framework for responsive front-end web development. It provides easy to use HTML components that include *buttons*, *dropdowns*, *navigation bars*, *form input field groups*, *icons* and others, out of the box CSS styling classes and JavaScript for the creation of *modal* windows, animations and event handling. One important thing to note about Bootstrap is that the library depends on the *jQuery* library which means that both need to be included in order to harness the full power of Bootstrap and use the available JavaScript functions.

3.2.6 Google Code Prettify

Google Code Prettify[33] is a JavaScript library for syntax highlighting of source code that is embedded in HTML. It supports all C-like, Bash-like and XML-like languages and does not required the user to specify the used language, meaning that the library is able to determine it on its own. The recognized languages include C, JavaScript, Java and Python. The library provides a couple of code styling themes that the users can choose from, additionally enabling customization via CSS. In order to make use of Google Code Prettify the user needs to provide the *prettyprint* CSS class in a HTML *<pre>* or *<code>* tag and the code included in those HTML elements will be highlighted based on the selected CSS style. The library begins execution when the HTML page fires the *onload* event that signals the page has completely loaded all content, including images, script files and CSS files. Additionally, it provides the user with the ability to execute "prettyprint" programmatically by calling *PR.prettyPrint();* inside JavaScript. This can be useful if additional text is added to the view dynamically and it needs to be highlighted after the onload event has been fired by the web page.

3.3 Graph-Objects Creation

This subsection provides a detailed explanation on how the CFA and ARG graph objects are created using the *D3* and *Dagre-D3* JavaScript libraries presented in Subsec-

[31] `http://api.jquery.com/`
[32] `https://getbootstrap.com/docs/3.3/`
[33] `https://github.com/google/code-prettify`

Listing 6: Graph Creation with Dagre-D3

```
1.  function createGraph() {
2.    var g = new dagreD3.graphlib.Graph().setGraph({}).setDefaultEdgeLabel(
3.       function() {
4.          return {};
5.        });
6.     return g;
7.   }
```

tions 3.2.1 and 3.2.2. The graph objects creation begins when the web workers are spawned and the main script sends the CFA and ARG JSON data using the self-written communication protocol presented in Section 2.2.2. Details regarding the message handling are discussed in Subsection 3.4. The first step in the graph creation is creating an *empty* graph object using the Dagre-D3 library as displayed in Listing 6. The displayed function, *createGraph()*, generates an empty graph object, note the call *setGraph({})* in line 2. The additional function call *setDefaultEdgeLabel()* ensures that edges without labels added to the graph will stay blank. Dagre-D3 also provides a range of configuration options for the graph object. For example the *rankdir* property can be used to determine the direction for node ranking. The possible values are top-to-bottom (TB), bottom-to-top (BT), left-to-right (LR) and right-to-left (RL). For the creation of the CFA and ARG graphs the default setting (TB) is used which means that nodes with higher rank will be placed below or further down as opposed to nodes with lower rank. Additional configuration properties include *nodesep* with default value 50 and *edgesep* with default value 10 which determines the number of pixels that separate the nodes and edges horizontally and are kept as defaulted. The *ranksep* property can be used to define a number of pixels between each rank in the layout, which is defaulted to 50. The full list of options is included in the Dagre[34] wiki page.

Once the empty graph object is created it needs to be populated. The Dagre-D3 library provides convenient methods to add nodes and edges to the graph object. Nodes are appended by using the *setNode(arg1, arg2)* method that is called on a graph object and receives two parameters. The first parameter is an unique identifier for the node that will be used internally by Dagre-D3. If the user sets several nodes with the same unique identifier, only the last node will be added to the graph and the previous ones will be ignored by the layouting algorithm which can be compared to inserting the same key in a hash table. The second argument passed to the function is a JavaScript object containing key-value pairs. The keys include *label* that is used for creating the node's label, *labelStyle* that is used to define a CSS style for the node's label, *class* that is used to provide a CSS class to the created node, *id* that is used to provide an unique identifier to the node that can be used later during DOM traversal to select the node, *shape* that is used to specify the node's shape which can either be one of the shapes provided by the library, in example *rect*, *circle*, *ellipse* or *diamond*, or a custom shape and *style* that is used to specify the node's CSS style. As values to the listed key properties the users can pass a callback function which enables differentiation based on the currently processed node data, thereby setting for example different shapes based on node *type*. Edges are added to the graph using the Dagre-D3 function *setEdge(arg1, arg2, arg3)* and passing three arguments to it. The first parameter is the *source* of the edge and the second parameter is the *target*. The source and target of an edge are essentially nodes in the graph, therefore the values for the first two parameters

[34] https://github.com/cpettitt/dagre/wiki#an-example-layout

must be a node's unique identifier as provided in the first argument of the setNode(arg1, arg2) function. We are using the default graph ranking which is top-to-bottom and this means that the layouter uses the source and target edge values to determine the position of the nodes inside the graph. If a node has only outgoing edges, it will receive the rank one and if a target node has an incoming edge from a source node that has, for example the rank two the target node will receive the rank three. The third parameter passed to the function setEdge(arg1, arg2, arg3) is an object that determines the appearance of the edge. The keys in said object include *label*, *labelStyle*, *id* and *class* that have the same purpose as the ones used in the setNode(arg1, arg2) function. Additionally, the edge configuration properties include *lineInterpolate* that is used to set the interpolation mode of the edge based on the *D3 interpolation options*[35] and *weight* that is used to assign a weight value to the edge where higher weight edges are made shorter and straighter in comparison to lower weight edges.

One drawback presented by the Dagre-D3 library is that it focuses on handling medium-sized graphs limited at around 1000 contained nodes. Due to the fact that for large programs the data provided by CPAchecker, especially for ARG graphs, often contains more, a special solution needed to be implemented. If the amount of nodes exceeds a predefined value, currently set at 700, the graph data will be split in multiple graph objects containing special nodes and edges to visualize the cohesion between the graphs as displayed in Figure 10 which are explained in Subsection 4.3.1.

3.4 Web Workers for Result Visualization

In order to increase performance the creation of the CFA and ARG graph objects was moved to background-running threads using web workers. The main script spawns, in most cases, two workers, the CFA worker and the ARG worker. If the CPAchecker analysis is not based on *abstract states* there is no ARG data available, so in this case the ARG worker is also not spawned by the main script. Since CPAchecker is a locally run software and the goal is to be able to view the generated report in any browser, it was not possible to store the web workers code in separate files due to security restrictions in the browsers Google Chrome and Chromium as mentioned in Subsection 2.2.1. Listing 7 displays the workaround code used in the JavaScript template used for the report generation. The variables *cfaWorker* and *argWorker* denote the web worker objects. Both functions, *cfaWorker_function* and *argWorker_function*, contain the logic used to create the CFA and ARG graph objects. The code contained in the functions is converted to a string, note the *toString()* method call, and enriched with a leading *"("* and a trailing *")()"* which makes the functions self-invoking. The, thereby, created string is passed as an argument to the *Blob* constructor which essentially creates an in-memory file containing the self-invoking function. The *uniform resource locator* (URL) to this file is then passed to the worker constructor, which spawns the actual worker as shown in lines 1 and 4 in Listing 7. The conditional check in line 3 ensures that if ARG data, as described in Listing 2, is not provided by CPAchecker the web worker is also not created in order to prevent unnecessary memory usage.

After spawning the workers the main script adds the required event listeners to the worker objects in order to enable the message transfer between it and the workers. There are two event listeners that the main script adds to each worker. The first event listener added to the *cfaWorker* is a *cfaWorker.addEventListener("message", function(m) {...}* which listens for the *message* event that is fired when the worker sends a "message" to the main script. The parameter m passed to the callback function for the "message" event is the message object. The property *data* of the message object carries the actual message send from the worker.

[35] `https://github.com/D3/D3-3.x-api-reference/blob/master/SVG-Shapes.md#line_interpolate`

Listing 7: Web Workers in CPAchecker report

```
1. var cfaWorker = new Worker(URL.createObjectURL(new
    Blob(["("+cfaWorker_function.toString()+")()"], {type: 'text/javascript'})));
2.    var argWorker;
3. if (argJson.nodes) {
4.   argWorker = new Worker(URL.createObjectURL(new
    Blob(["("+argWorker_function.toString()+")()"], {type: "text/javascript"})));
5. }
```

The function then examines the message received and reacts accordingly. The second event listener that the main script adds to the cfaWorker is *cfaWorker.addEventListener("error", function(e) {...}* which listens for the occurance of an *error* event that is fired by the worker when an error is thrown during code processing. Similar event listeners are added to the *argWorker* object only if the worker is created which is not the case if the CPAchecker verification analysis is not based on abstract states.

Figure 5 displays the communication protocol used for data transfer between the main script and the workers, where the right-hand side of the figure displays the message exchange between the main script and the *cfaWorker* and the left-hand side displays the communication between the main script and the *argWorker*. Since the communication is bidirectional the CFA worker and the ARG worker need to register an event listener on the "message" event as well, as described in Section 2.2.1 covering the general use of web workers. The listeners are added by *self.addEventListener('message', function(m) {...}* which is similar to the one added on the worker object by the main script with the difference that it is added to the worker itself, note the key word *self*. The "data" property of the "message" object can only contain strings, therefore in order to transfer an object one must use the *JSON.stringify()*[36] function which converts a JavaScript value in a JSON string. The first message that the main script sends is directed to the CFA worker and contains the key "json" and the CFA JSON data displayed in Listing 1. The second message the main script sends is directed to the ARG worker, marked with the number two in figure 5, containing the key "errorPath" and the error path data contained in the CFA JSON object. It is important to note that this message is not always send to the argWorker. If there is no counterexample found by CPAchecker during the verification run, there is no error path data available. The third message that the main script sends is also directed to the ARG worker and contains the key "json" and the ARG JSON data displayed in Listing 2. Upon receiving those messages the workers begin with the graph objects creation respectively for the CFA and ARG graphs. The CFA worker creates one graph object per each function included in the source code beginning with the "main" function or if there is no function called "main" it beginns with the first function contained in the *functionNames* array from the CFA JSON object following the logic explained in Subsection 3.3. During the CFA graph creation there is an important point that the graph creation logic considers which is the *combinedNodes*. A combined node is a node that contains linear sequences of "normal" edges (statement-edges, declaration-edges, and blank edges). While iterating over the *nodes* data the CFA worker checks wheter or not the currently processed node is a part of the *mergedNodes* which contains information about the nodes that are part of a combined node. If this is the case the node will not be further processed, thus not setting it in the graph

[36] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON/stringify`

object. Otherwise the CFA worker will further check if the node is contained in the keys of the "combinedNodes" object from the CFA JSON data, as shown in Listing 1. If this is the case, it will be set with the corresponding label from the *combinedNodesLabels* object, thus creating the "combined node" as can be seen in Figure 8 which displays a counterexample report generated with the solution presented in this thesis. Once the graph object for the first function is created it will be send automatically to the main script and the worker will continue to generate and store the further required graph objects. The main script in turn takes care of the rendering because as explained in Subsection 2.2.1 the web worker does not have access to the DOM. The message used in this case is denoted with the number 4 in Figure 5 where the value behind the key "graph" is the graph object converted to JSON string, the "id" is a sequential number beginning with 0 and "func" stands for the function name for which the graph was created. Upon receiving the message the main script must create a graph object using the provided JSON string containing the graph data. This is achieved by creating an empty graph object as show in Listing 6 and assigning the data to it using the JavaScript standard function *Object.assign(g, JSON.parse(m.data.graph))* where the first parameter passed stands for the empty graph object and the second is the parsed JSON string graph object send by the worker. Furthermore, the main script dynamically adds an SVG element to the HTML at the correct position of the report HTML using the D3 library. The actual rendering is performed by the function call *render(D3.select("#cfa-svg-" + id + " g"), g);* where *render* is a renderer object created with Dagre-D3, *var render = new dagreD3.render();*, and the parameters passed to the function are the created SVG element which is selected with the D3 library using its unique identifier (id) and the created graph object. After the rendering is finished the main script will send the message *"renderer":"ready"* to the CFA worker telling it that the main script is ready to receive and render further graph objects. The CFA worker in turn will either respond with a similar message as before, only containing the "graph" data, "id" and "func" for the next graph or, if all CFA graphs are already send to the main script, the worker will respond with the message *"status":"done"*. As mentioned above the CFA worker generates one graph per function included in the source code but if the amount of nodes for a given function exceeds the defined limit of 700 nodes the worker will split those nodes in multiple graphs due to the restriction in the Dagre-D3 library, as shown in Subsection 4.3.1. Upon receiving the "status":"done" message from the CFA worker the main script will send the message "renderer":"ready" to the ARG worker. This tells the ARG worker that the main script is ready to receive and render the ARG graphs which triggers the message with the keys "graph" and "id", marked with the number 7 in Figure 5, where the values are the JSON string representation of the ARG graph object and a sequentially increased number starting with 0. The main script will handle this message similarly to handling the graph objects send by the CFA worker. It will create an empty graph object, parse and assign it the data contained in the message, it will create an SVG element at the correct position in the report and use the Dagre-D3 render method to create the viewable graph. Following this the main script will again send the "renderer":"ready" message to the ARG worker. Upon receiving the message the worker will either send additional ARG graphs, if available, or will send the message "status":"done" to the main script. As mentioned above the only reason to have multiple ARG graphs is if the amount of nodes exceeds 700 which will cause the worker to split those due to the restriction in Dagre-D3. Upon receiving the "status":"done" message from the ARG worker the graph generation is finalized and the report can be viewed and interacted with by the user.

3.5 Data Binding in CPAchecker Report

The previous solution was highly dependant on the *AngularJS* library which can be seen in figure 4.10: *The controllers and their communication* on page 30 of the *Towards Understandable CPAchecker Counterexamples thesis* [6]. The complicated communication between the controllers achieved by the use of Angular's *$broadcast* events is not only challenging for debugging in case of an error but it also affects the performance because of the mutliple event listeners that are internally added by the library. The new solution implements the event handling using D3 which makes the communication between the AngularJS controllers obsolete, thereby reducing the complexity to the one displayed in Figure 6. Figure 6 displays the AngularJS controllers used in the CPAchecker report generation and provides information about their *$scope* variables, in the first segment below each controller's name and their methods in the second segment. A detailed description of the features provided by the controllers is available in Subsection 4.3. The *ReportController* is, as in the previous solution, bound to the HTML body tag which makes it the parent of all other controllers as they are bound to HTML elements inside the body. It takes care of general tasks like keeping track of the currently selected *tab*, the displayed *logo* and the information displayed in the *help buttons*. Additionally, the ReportController provides functions to change the current tab - *setTab(tabIndex)*, check if a given tab is set *tabIsSet(tab)* and retrieve the currently selected tab index - *getTabSet()*. The *ErrorpathController* holds the *errorPath* data and handles the error-path table tasks which enable the traversal of the error path and the highlighting of the graph or source code element by invoking the functions *errPathPrevClicked($event)*, *errPathStartClicked()*, *errPathNextClicked($event)* to handle the corresponding button clicks and the function *clickedErrpathElement($event)* which handles a click on a row in the error-path table. The *SearchController* provides the search functionality connected to the error-path table which can be used to examine a variable's initialization and value changes by using the function *searchFor()*, additionally providing a listener for the enter-key pressed event - *checkIfEnter()* that will also trigger the search. The *ValueAssignmentsController* displays the popup window representing the current program state in an error-path table row by executing the function *showValues()*. The *CFAToolbarController* provides the features connected to the CFA graphs to the users. It enables the altering between the currently displayed CFA graph - *setCfaFunction()*, the zoom possibility - *zoomControl()* and the *redraw()* option which enables the user to define a different graph split ratio. The *ARGToolbarController* is similar to the CFAToolbarController in an essence that it provides similar interaction capabilities to the user. It enables the switch between the displayed *complete* ARG graph and an ARG graph that consists only of the error-path elements - *error-path* ARG, zoom functionality similar to the one available for the CFA graphs and a redraw option that allows the users to recreate the ARG graph using a different graph split ratio. The *SourceController* handles the altering between the displayed source file in the *Source* tab - *setSourceFile(value)*. As displayed in Figure 6 a high amount of functions were reused to an extend from the previous solution. The important thing to note is that there is no communication between the controllers which also means there are no dependencies between them.

3.6 Library Interchangeability

The implementation and architecture set up of the new solution for the CPAchecker report generation uses multiple third-party libraries, as listed in Subsection 3.2 but each one is taking care of a specific task without highly depending on the others, with small exceptions. This was one of the persuaded goals during the implementation in order to

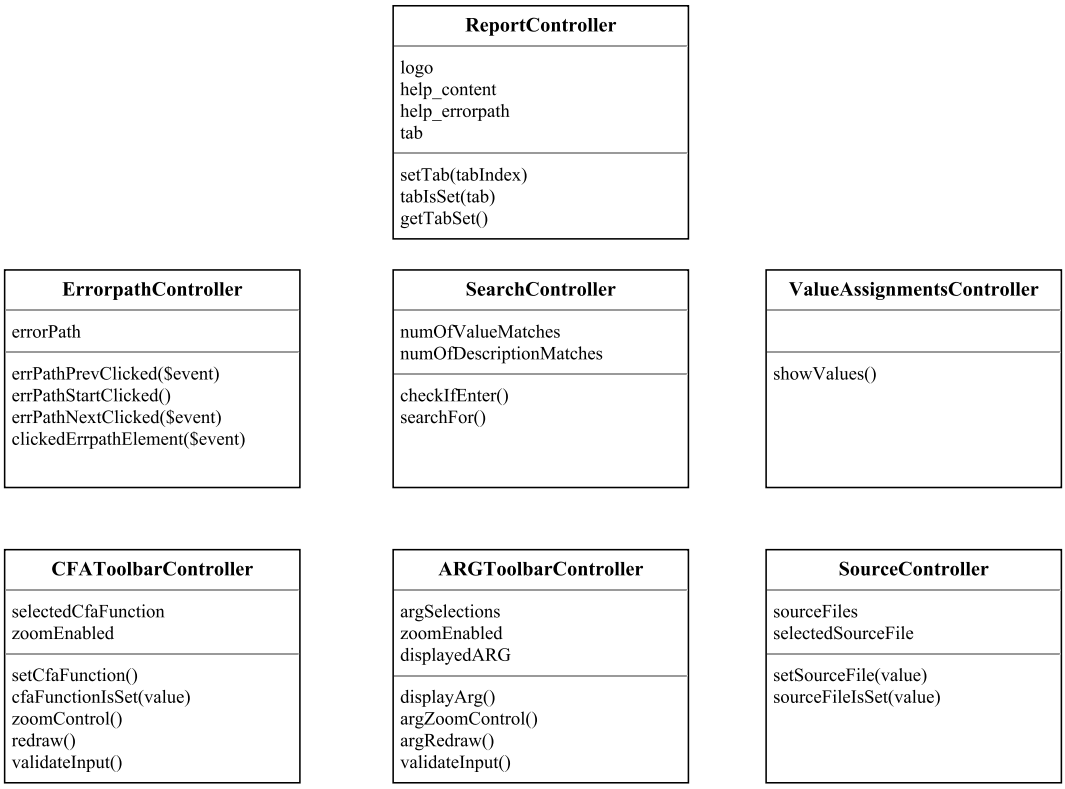

Figure 6: AngularJS Controller in CPAchecker Report

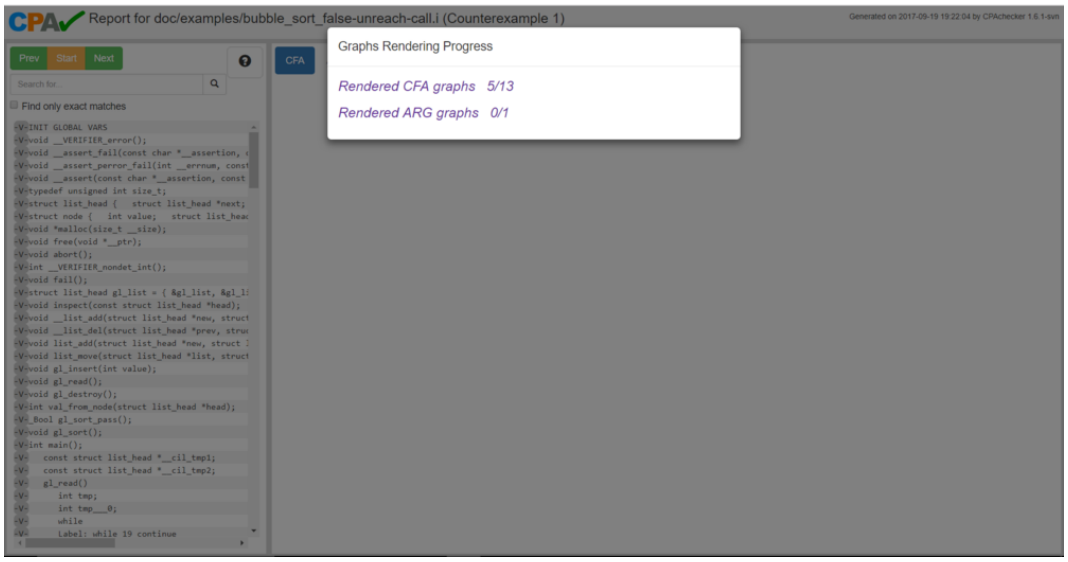

Figure 7: User Feedback Modal Window

allow library interchangeability. By removing the communication between the AngularJS controllers it will be possible to remove the need of the library completely, in example by moving the tasks handled by the controllers in native JavaScript or jQuery and Bootstrap. The Dagre-D3 library takes care of the graph objects creation and the relevant code is almost exclusively contained in the web workers. This means that if another library is found, or an own solution for graph creation and rendering is build, that handles the graph creation better in regards of performance or usability, it can be embedded in the CPAchecker generated report without much effort.

4 Verification Report

This section of the document focuses on the product of a CPAchecker verification run - the generated report that can be viewed by the users. Subsection 4.1 provides an overview of a custom solution that informs the CPAchecker users about the progress of the CFA and ARG graph rendering when the report is being opened. Subsection 4.2 focuses on the page layout of the generated report. All available interaction features are presented in detail in Subsection 4.3 and an overview of the problems remaining in the solution presented in this thesis is contained in Subsection 4.4.

4.1 User Feedback during Graph Creation

As mentioned in Subsection 3.4 the graph rendering is performed by the main script because the workers do not have access to the DOM and it is a costly task that "freezes" the browser which is a drawback in regards of usability because the user is not aware of what exactly is happening at the moment. In order to provide some feedback to the user during graph rendering a bootstrap *modal*[37] window was introduced, displayed in Figure 7. A modal window is a dialog box that is displayed on top of the current page. As shown in

[37] `https://getbootstrap.com/docs/3.3/javascript/#modals`

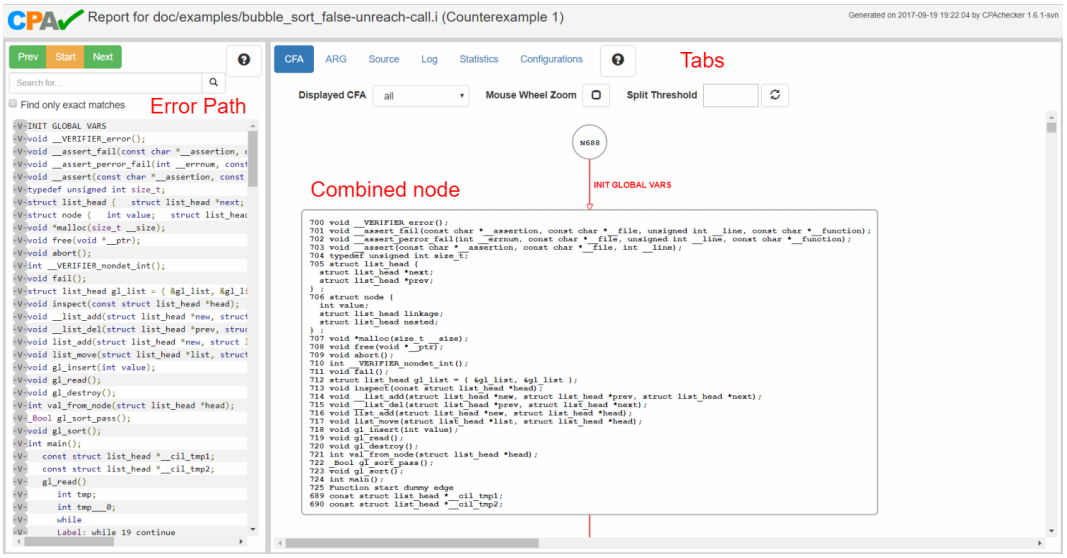

Figure 8: Counterexample Report - New Solution

Figure 7 the modal window is highlighted in white to draw focus as the actual page below is grayed out. The modal window is displayed automatically when the report is opened in a browser. The header of the window displays the text *Graphs Rendering Progress* and the two lines in the body represent the current state of the CFA and ARG graph rendering. In the example displayed in Figure 7 the report contains 13 CFA graphs which can be seen in the line *Rendered CFA graphs 5/13* and up until this moment 5 of the 13 graphs have been rendered. The value before the slash (/) is updated dynamically every time the main script receives a message containing a CFA graph object and finishes rendering it as described in Subsection 3.4. This means that once the main script receives the message "status":"done" from the CFA worker, the displayed value will be *13/13*. The second line carries the same information in regards of the ARG graphs rendering. Usually a modal window will present the user with selection options, in example buttons to *close* the window or *save* the current state. Such options are not required for the generated report and instead the modal window is closed programmatically once all graphs have been successfully rendered in which case the user is presented with the view displayed in Figure 8 if a counterexample is found or with the view displayed in Figure 9 otherwise.

4.2 Report Page Layout

This sections focuses on the layout of the report page presented to the user once the CFA and ARG graphs have been rendered and the modal window introduced in Subsection 4.1 has been closed. Figure 8 displays the report starting point in case a counterexample is found by CPAchecker during the verification run. The page layout is the same as the one from the previous solution, as can be seen in Figure 1. On the left-hand side the page contains a table-like representation of the *error-path* data and the right side is divided into six tabs, the *CFA* tab which contains the CFA graphs and is selected by default, the *ARG* tab which contains the ARG graphs, the *Source* tab which contains a table like representation of the source code of the program that was verified with CPAchecker, the *Log* tab which presents the generated log output from CPAchecker, the *Statistics* tab displaying the statis-

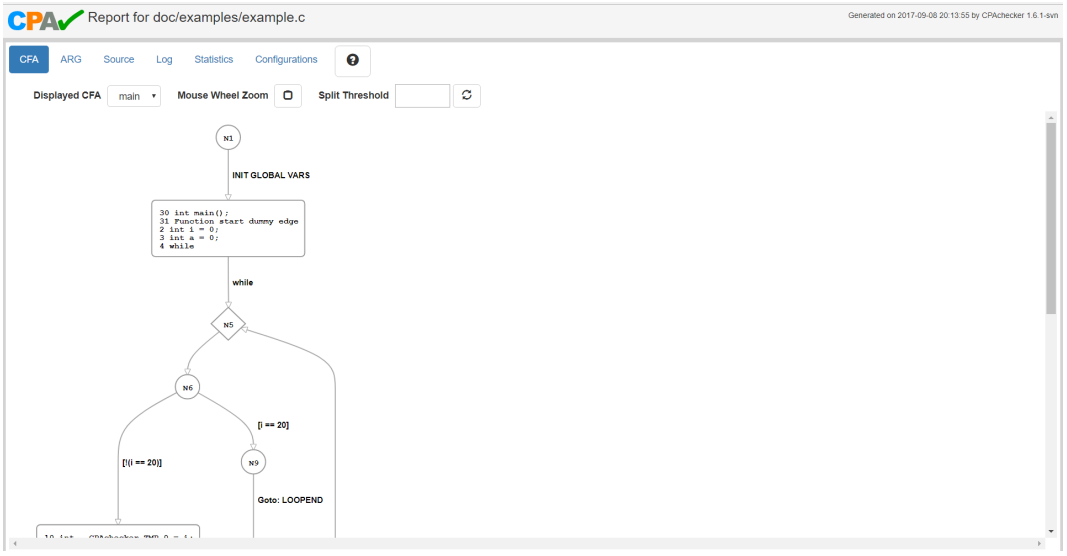

Figure 9: Report - New Solution

tics information provided by CPAchecker which contains, amongst other things, the time required for the verification and the used memory and the last tab *Configuration* which displays the configuration options passed to CPAchecker at the start of the verificaition run. The differences in comparison to the previous solution include the way the CFA and ARG graphs are generated and the toolbars contained in those tabs which are discussed in Subsection 4.3. By dividing the page in different tabs it is ensured that only one piece of relevant information is displayed to the user at a given time, thereby keeping the page clean. The error-path table on the left-hand side of the page is only displayed if the CPAchecker verification run has found a counterexample for the verified source code. In this case the error-path table takes 25% of the total page width, leaving the remaining 75% for the tabs section. If no counterexample is found, the tabs section is displayed in the whole page as can be seen in Figure 9.

4.3 Report Features

This part of the document focuses on the available interaction capabilities provided to the user, for which a brief overview is included in the *Report.md*[38] file contained in the CPAchecker GitHub repository. Furthermore, the generated report contains two *Help buttons* that provide additional information about the generated graphs and the available features, marked with a question mark (?), as can be seen in Figure 8. One of the buttons is contained in the error-path part of the report and provides information about the available features in the error-path table. This button is only available if a counterexample is found during the CPAchecker verification run. The second help button is positioned to the right of the tab names and is always available independent of the verification outcome. Those buttons provide information about the data included in each tab as well as the interactive capabilities provided to the user. All available features in the new report, generated by CPAchecker, are explained in detail in the following subsections.

[38] `https://github.com/sosy-lab/cpachecker/blob/trunk/doc/Report.md`

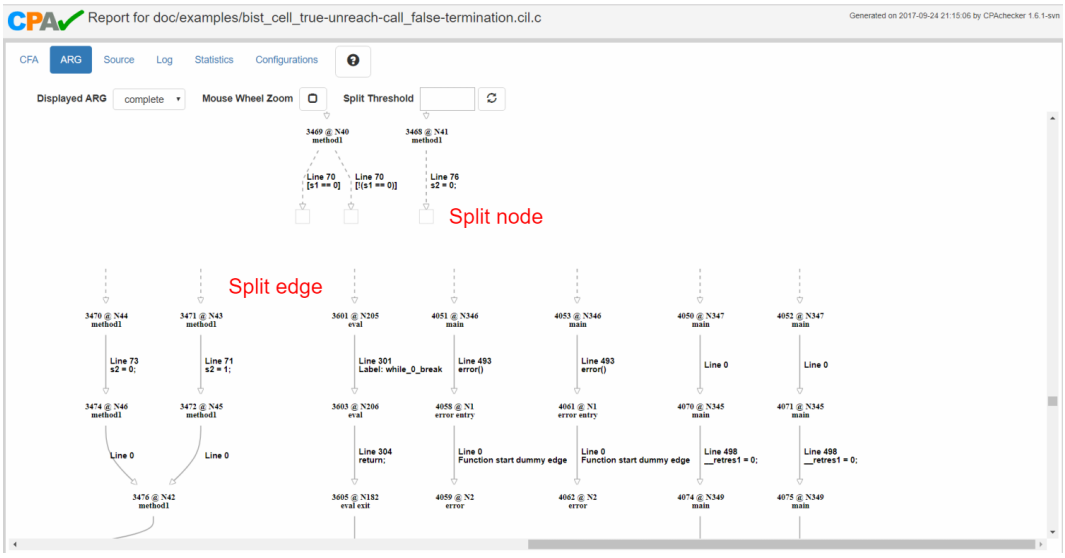

Figure 10: Artificial Graph Split

4.3.1 Graph Split

This subsection focuses on a functionality that results from the Dagre-D3 restriction on graphs containing more than 1000 nodes as presented in Subsection 3.3. If the generated CFA and ARG graphs contain higher amount of nodes as the currently defined upper bound of 700 nodes, the graphs will be split. In order to keep the cohesion between the graphs that were artificially separated the implementation logic adds addtional nodes and edges to the graphs in order to make the users aware of the split as shown in Figure 10. The *split nodes and edges* shown in the Figure 10 act as an intermediate stop between the two graphs. If the split was not required by the technical restriction, the edge with label *Line 70 [s1 == 0]* would have been connecting the nodes with labels *3469 @ N40* and *3470 @ N44*. Additionally, if the user places the mouse pointer on one of the split nodes, a tooltip box will be displayed as shows in Figure 11. It provides the information that the node's type is a *place holder* which means that it is not an actual node generated during the CPAchecker verification run. In addition, the tooltip box displays information about a double-click event that will move the viewport to the node that was originally the target node. Extending the example from before, if the user double-clicks on the empty square pointed by the edge with label *Line 70 [s1 == 0]*, the node with label *3470 @ N44* will be marked as displayed in Figure 13. If this node was outside of the current viewport, the page will scroll to its location. The split edges are subjects to the same rules. If the user places the mouse pointer over an artificial edge, a tooltip box will be displayed as shown in Figure 12. The tooltip box displays the edge type as *place holder* following the same reason as for the split node, addtionally providing the information about the available double-click event which will scroll to and mark the original edge as shown in Figure 14.

4.3.2 Error-Path Features

If the CPAchecker verification run finds a counterexample for the verified software, it will generate a counterexample report as shown in Figure 8. The left-hand side of the

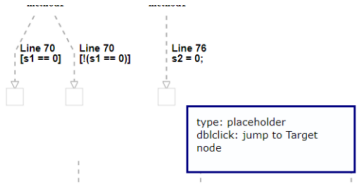

Figure 11: Split-Node Tooltip

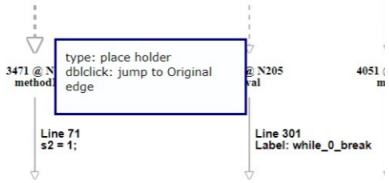

Figure 12: Split-Edge Tooltip

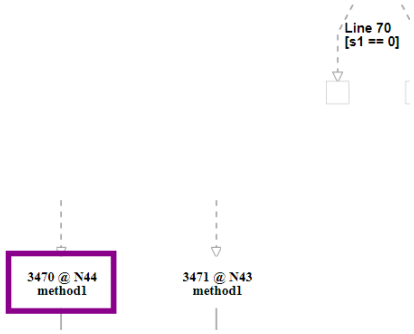

Figure 13: Split-Node Double Click

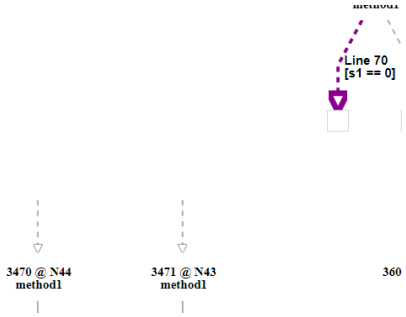

Figure 14: Split-Edge Double Click

generated report contains the *error-path window*. This section of the report includes a table representation of the source code lines that are a part of the error path leading to the specification violation where each table row contains a source code line. The included indentation represents the hight of the call stack. Additionally, code syntax highlighting is provided using the Google Code Prettify library introduced in Section 3.2. On the furthest left of each table row there is a button marked with a *-V-*. By clicking on this button the users will be shown a *popup window* that contains information about the program state at the point represented by the source code line displayed in the table row preceded by the clicked button as shown in Figure 15. The users can open multiple popup windows to view the information about the initialized variables and their values at the program state represented by the row in the error-path table. As can be seen in Figure 15 the clicked buttons are outlined in green and between the two points in the program execution, which are displayed, a variable called *___cil_tmp7* with the value *73U* was initialized which can be seen in the last line of the second popup window. In order to close the program state information popups the users can click again on the previously selected buttons.

The search bar above the error-path table provides the users with the possibility to search for program variables contained in the error path. When the users enter a variable name and press the search button to the side of the input field or the enter key a search in the error-path table will be executed. The search results will be displayed to the users in two lines below the search input field as shown in Figure 16. The *Matches in value-assignements (V)* value displays the amount of times the searched variable is included in the program states represented behind the -V- button which holds the information about the initialization of the variable and the number of times its value changes. The second line, *Matches in edge-description*, displays the amount of times the name of the variable occurs in the source code lines displayed in the error-path table. Furthermore, if the name of the searched variable is included in the program line displayed in the error table, the line will be highlighted in blue

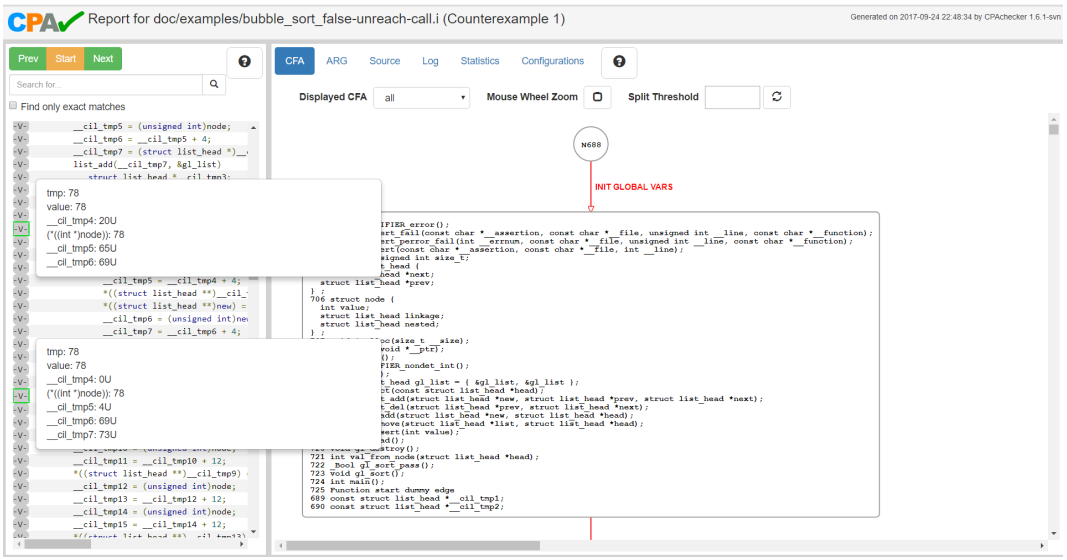

Figure 15: Current Program State

and if the variable was initialized in the same line or its value has changed, which will be shown in the program state popup opened with the -V- button, the line will be highlighted in green. Additionally, if both conditions occur, the line will be highlighted in a blue-green gradient color as shown in Figure 16. The *Find only exact matches* checkbox can be set by the users to limit the search to an exact match. The search example displayed in Figure 16 shows the search for a variable named *tmp* without setting the exact matches limitation which leads to every variable containing the search criteria, *tmp*, being considered during the search. This includes, in example, the variables called *__cil_tmp0*, *__cil_tmp4*, *__cil_tmp5* and *__cil_tmp6*. The values found for the displayed example are matches in value-assignments - 19 and matches in edge-description - 178. Figure 17 on the other hand displays the search results using the same search criteria but only considering exact matches which results in the variables *__cil_tmp0*, *__cil_tmp4*, *__cil_tmp5*, and so on being ignored by the search. Ultimatelly delivering the value for matches in value-assignments - 1 which means that after the initialization of the variable "tmp" its value was never changed and the matches in edge-description - 5 which means that the "tmp" variable occurs 5 times in the error-path table rows.

The error-path table in the generated counterexample report also provides the user with the capability to *walk along* the error path in the CFA and ARG graphs or the Source tab. If the user clicks on an error-path table row, the corresponding position will be marked in the currently opened tab on the right, if it is one of the CFA, ARG or Source tabs. If the currently displayed tab on the right is one of the Log, Statistics or Configurations tabs, the view will switch to the ARG tab and then mark the corresponding element. Figures 18 and 19 display the behaviour of the counterexample report if an error-path table line is clicked by the user and the currently selected tab on the right is the CFA tab. If the error-path element is displayed as an edge in the CFA graph, the edge and the error-path table row will be marked as shown in Figure 18. If the edge is a part of a CFA *combined node*, the error-path table element and the corresponding representation in the CFA graph will be highlighted as displayed in Figure 19. The combined node containing the error-path

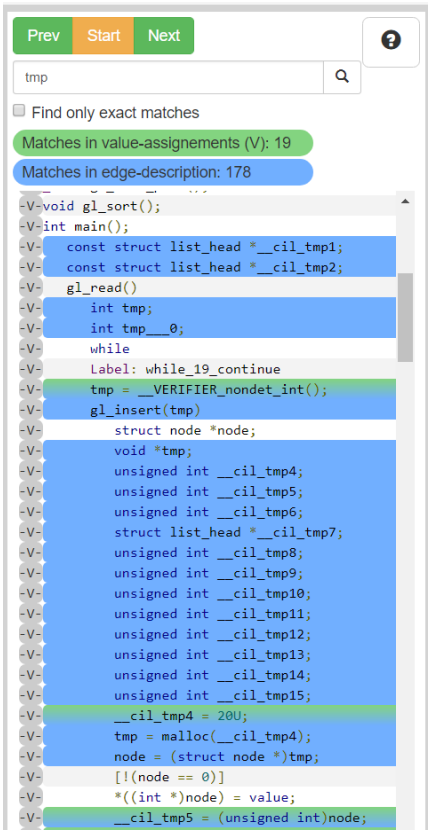

Figure 16: Error-Path Search Functionality

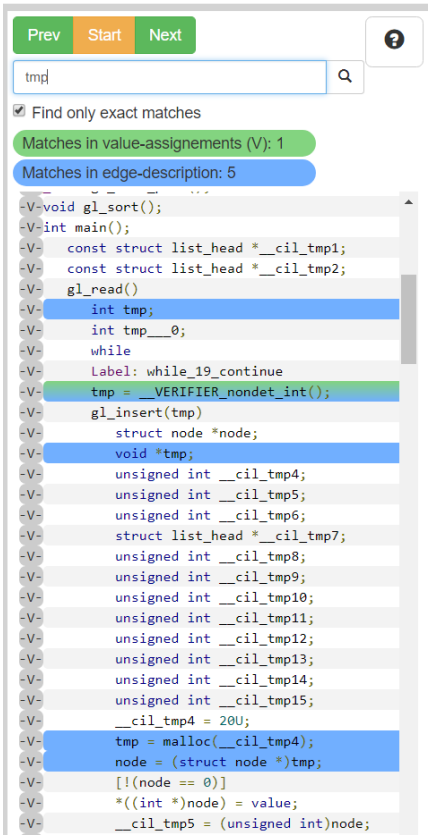

Figure 17: Error-Path Search Functionality - Exact Match

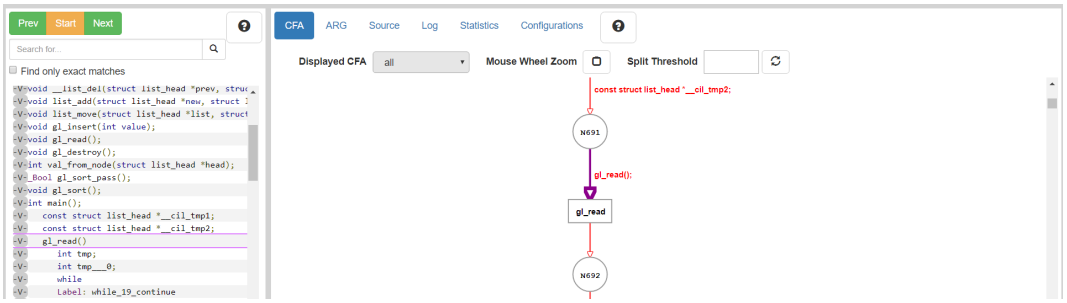

Figure 18: Error-Path Walk Along CFA Edge

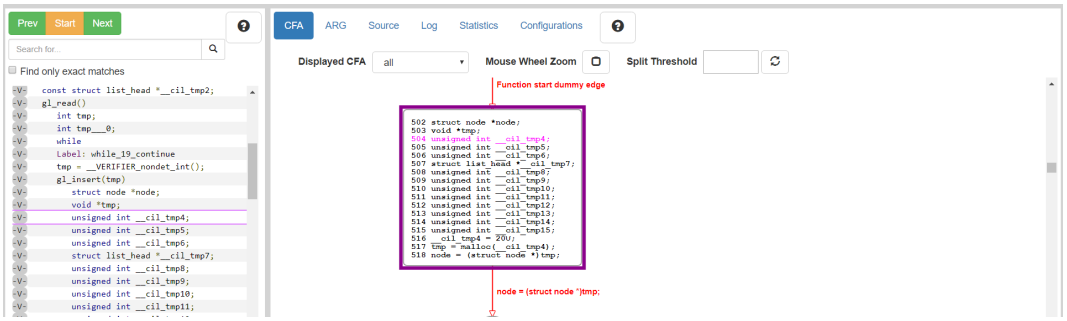

Figure 19: Error-Path Walk Along CFA Combined Node

element will be outlined and additionally the program source code statement matching the node label will be highlighted as well.

If the user marks an error-path element by clicking on it and the currently selected tab on the right part of the CPAchecker generated report is the ARG tab, the node corresponding to the error-path element will be marked in the graph as displayed in Figure 20. For some elements the CPAchecker verification run does not create an abstract state which means there is no corresponding node in the ARG and therefore a click on the error-path table row will not mark an ARG element. If the currently selected tab on the right is the Source tab and the user marks an error-path element from the table by clicking on it, the source code line corresponding to it will be marked as shown in Figure 21.

The *Prev*, *Start* and *Next* buttons available above the search bar in the error-path window provide aid to the user for the error-path walk along. Clicking the Start button will mark the first row in the error-path table and scroll to it, additionally highlighting the corresponding element in the right part of the report applying the same logic used when the user selects an error-path table element by clicking on it. The Prev and Next buttons can be used to navigate back and forth in the error path. They will highlight the previous or next row in the table on the left and the corresponding element in the currently active tab on the right as explained above and shown in figures 18, 19, 20 and 21. The Prev and Next buttons can also be used directly after manually selecting an error-path element from the table by clicking on it. This allows the user to select a starting point in the program's error path for further navigation.

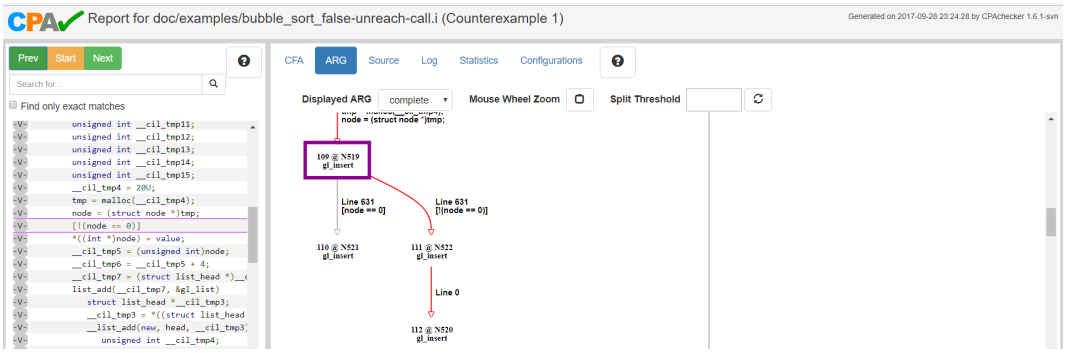

Figure 20: Error-Path Walk Along ARG

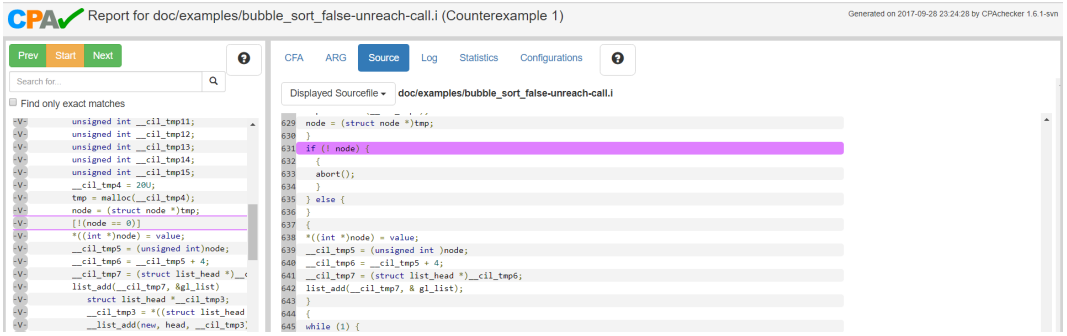

Figure 21: Error-Path Walk Along Source

Figure 22: CFA Toolbar

4.3.3 CFA-Tab Features

There are multiple interactive features available to the user in the CFA tab of the generated report. Positioning the mouse cursor over a graph element, an edge or a node, will display an information tooltip providing details for the element. A tooltip for a *normal node*, displayed as a circle in the CFA graphs, includes information about the function to which the node belongs and its *reverse postorder id* which represents the position of the node in the *queue* used by CPAchecker during the verification run. In addition to those details a tooltip for a *combined node* will include the unique node identifiers combined within it. The *function call nodes* in the CFA graphs stand for functions called within other functions and are represented by rectangles. The tooltip shown for those nodes includes the type of the node which is "function call node" and the double-click event available to the user. The double-click event will select the function represented by the function call node, which is shown in its label, and display only the CFA graph that represents it. The tooltip shown when the mouse cursor is positioned over an edge in a CFA graph displays information about the double-click event available, which will set the Source tab as currently active tab on the right and will mark the source code line corresponding to the clicked edge similar to the highlighting show in Figure 21, additionally scrolling to the DOM element's position to present it to the user. The CFA *toolbar*, displayed in Figure 22, provides further interactive options to the user. The dropdown list on the left provides the ability to select which CFA graph is currently displayed. The default value is "all" which means that all CFA graphs will be shown to the user, one below the other, beginning with the CFA for the "main" function. The available values in the dropdown list represent the names of the functions included in the verified program because one CFA graph is created for each function. Per default the mouse wheel is used to scroll vertically in the CFA tab. Additionally, the user can *drag* a CFA graph by clicking and holding the left mouse button near it and moving the mouse around which is known as a *pan* event. By selecting the *Mouse Wheel Zoom* checkbox the user can use the mouse wheel to zoom in and out of the graph. The zoom functionality is applied separately to each SVG element that contains a CFA graph. This provides the possibility to apply different zoom levels to each graph. The *Split Threshold* input field and the connected *redraw* button can be used if a CFA graph is artificially split due to the restricition in the Dagre-D3 library. It allows the user to set a different graph split threshold as the default value of 700 nodes. The input is validated to ensure that only numbers between 500 and 900 can be entered and will display an alert if the condition is violated. By clicking the redraw button the main script will send a *"split"* message to the CFA web worker containing the entered value as displayed with number 17 in Figure 5. This will cause the recreation of all CFA graph objects using the new split threshold and sending them one by one back to the main script for rendering. During the communication between the main script and the CFA worker the user will be presented once again with the modal window shown in Figure 7.

4.3.4 ARG-Tab Features

The ARG tab of the generated report provides similar interaction features to the user as the CFA tab. By positioning the mouse cursor over an ARG node a tooltip will be

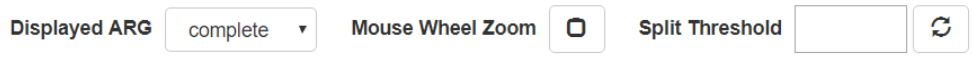

Figure 23: ARG Toolbar

presented. The tooltip includes information about the *function* to which the node belongs, the *type* of the node and the available double-click event. The node type is only displayed for special nodes that are also marked with different colors in the graph. Blue for a *highlighted* node which represents an important node depending on the used analysis. Green for a *covered* node which stands for an abstract state that was covered by another abstract state. Orange for a *not expanded* node which represents an abstract state that is not processed by the analysis and red for a *target* node which denotes an abstract state that will violate the used specification if reached. The double-click event available on each ARG node will set the CFA tab as the currently active tab and will highlight the corresponding CFA node by marking it similarly to the solution displayed in Figure 19. By positioning the mouse cursor over an ARG edge a tooltip will be displayed that contains information about the edge type. Additionally, the ARG tab also contains the *pan* event which provides the user with the possibility to move the graph around inside its SVG element. The ARG *toolbar*, displayed in Figure 23, provides similar initeraction options to the user as the CFA toolbar. The *Displayed ARG* dropdown list provides the user with the option to display the *complete* ARG graph which is the default setting or the *error-path* ARG graph which is an ARG graph build only with the elements contained in the error path. If the user selects the error-path ARG graph, the main script will send a *"errorGraph"* message to the ARG Web Worker as marked with the number 11 in Figure 5. The ARG worker in turn will send the graph object and the main script will process it. During the graph rendering the user will be presented again with the modal window shown in Figure 7. The *Mouse Wheel Zoom* checkbox and the *Split Threshold* input field with the redraw button trigger the same functionalities as explained previously for the CFA toolbar. They allow the users to zoom in and out of the graph and to redraw the ARG graph using a different split threshold than the defined value of 700 nodes.

4.3.5 Features in Other Tabs

The Source, Log, Statistics and Configurations tabs contained in the generated report are kept from the previous solution without any adjustments. The Source tab contains a table representation of the source code honoring indentations and providing syntax highlighting. Furthermore, the Source tab includes a dropdown selection menu that enables the user to switch between the currently displayed file, as can be seen in Figure 21. The Log tab provides a textual representation of the generated log output by CPAchecker during the verification run. The Statistics tab contains a textual representation of statistics generated by CPAchecker that include the amount of time taken by CPAchecker for the different verification steps and the used memory. The Configurations tab displays the information about the used configuration options that include the type of the used analysis and specification at the start of the verficiation run.

4.4 Remaining Problems

Following the implementation approach described in Section 3 most of the known issues contained in the previous report generation solution were resolved. However two

problems remain as described in the CPAchecker online documentation[39].

Error-path element highlighting in CFA tab. If a specific function is displayed by selecting it in the dropdown list contained in the CFA toolbar and the user selects an error path element from the table, either by clicking it or by using one of the navigation buttons, the element will not be shown to the user if it is contained in a different function as the one currently selected. As a workaround for this problem the report user can choose the value "all" in the dropdown menu in order to display all CFA graphs and select the error-path element again which will highlight the corresponding graph element accordingly, as shown in Figure 18 and Figure 19.

Error-path element selection only affects currently active tab. If the user selects an error-path element from the table, either by clicking it or by using one of the navigation buttons, only the active tab on the right will be updated following the rules presented in Subsection 4.3.2. In order to bypass this problem the user can switch the active tab manually, using the navigation bar at the top of the page, and then reselect the error-path element from the table.

Even though the listed problems remain in the generated report, the majority of the issues contained in the previous version, as listed in Subsection 1.1.1, have been eliminated by the used architectural and implementational approach.

5 Evaluation

This section of the thesis describes the conducted evaluation of the new solution for the CPAchecker generated report. Subsection 5.1 describes the taken evaluation approach. Subsection 5.2 provides an overview of the participants feedback gathered during the evaluation and the resulting implementation changes.

5.1 Evaluation Approach

The conducted evaluation consists of two parts, a *questionnaire* presented in Section A of the appendix and *user tasks* included in Section B of the appendix. The user tasks are specifically tailored assignments for the evaluation participants ensuring that all available functionalities in the generated report, described in Section 4.3, will be used, thereby making the users familiar with the new solution and providing them the possibility to evaluate those report-interaction features. The idea is to begin with the questionnaire, gathering some general information about the participant, execute the first defined task, marked with number 2 in Section A, and continue with the evaluation once the task is completed and the participant is familiar with the features of the new report. The first user task defined in Section B requires the user to obtain a file from the CPAchecker repository and execute a specific verification analysis and specification. The generated counterexample report must then be analyzed using the available features described in Subsection 4.3 which will assist the participant to find the cause of the error in the program. Once the first task is completed and the feature-related questions, marked with the number 3 in the questionnaire, are answered the participant is prompted to complete the second user task before continuing with sections 5 and 6 from the questionnaire. The second user task uses a different program that is analyzed by CPAchecker. There is no counterexample found which means that the

[39] `https://github.com/sosy-lab/cpachecker/blob/trunk/doc/Report.md`

Rating question	Answers
"Jump to Source" event	50% highest mark 50% second highest mark
"Jump to CFA" event	100% second lowest mark[40]
Error-path ARG graph	50% highest mark 50% distributed[41]
Error-path search feature	50% highest mark 50% second highest mark
Error-path walk along	50% second highest mark 50% distributed[41]
Performance improvement	25% highest mark 75% second highest mark
Usability improvement	50% second highest mark 50% distributed[41]
Functionality improvement	50% second highest mark 50% third highest mark

Table 1: Results Overview

resulting report will not contain an error-path table, thereby making the user familiar with the page layout displayed in Figure 9. Additionally, the ARG graph for the second program exceeds the split threshold which is then used to make the participants familiar with the feature connected to split graphs, displayed in Figure 10.

5.2 Evaluation Results

This subsection of the thesis provides an overview of the results from the feature, performance, usability and functionality rating questions as well as the implementation changes that resulted from the conducted evaluation. The gathered feedback was analyzed and the summary of the rating questions is contained in Table 1. Additionally, all available suggestions from the attendants were considered and the changes in the implementation, listed below, reflect the identical suggestions between multiple evaluation participants.

As we can see from the summary in Table 1 the evaluation attendants all agree, to a different degree that the new approach for generating the verification-run report is an improvement considering performance, usability and functionality in comparison to the previous solution. Additionally, all participants agree with the general layout of the report page and the used colors and shapes in the generated CFA and ARG graphs.

The following part of this subsection lists all implementation changes that resulted from the suggestions of the participants in the conducted evaluation.

Pan event. The ability to move a CFA or ARG graph around in a *drag and drop* matter was initially only available after the *mouse wheel zoom* checkbox was set in the respective toolbar. As per the wish of the evaluation participants the *pan* possibility is now available per default as well as after selecting the checkbox. Additionally, since the pan event is

[39] A bug in the "Jump to CFA" event was found during the evaluation.
[39] Although distributed, the ranking was never below the third highest mark.

added to the SVG element that contains the graph, for small graphs the resulting parent SVG element was narrow. This led to the confusion that the pan is not working but the reason was it was attempted outside the SVG element. This was resolved by matching the SVG width for small graphs to the page width.

Jump to CFA node from ARG. During the evaluation an issue with the generated report was found by the participants. When executing the available double-click event on ARG nodes the active tab was set correctly to the CFA tab and the corresponding element was highlighted as expected but the viewport was not focused correctly. This lead to the user not being able to see the highlighted element. As described in Subsection 4.3 the "jump" event from ARG node to CFA node is now working as initially desired.

Vertical scroll in the error-path table. Initially when navigating through the error path using the "prev" and "next" buttons there was no automatic vertical scroll available for the error-path table. This meant that the corresponding element in the active tab on the right was focused and highlighted but the error-path table row on the left was not visible. As suggested by all participants automatic vertical scroll in the error-path table was made available.

Mark labels in combined nodes during error path walk along. In the previous version of the new report-generation solution only the combined node was highlighted when an error path element represented by it was marked in the error-path table. This matched the logic contained in the previous solution. The majority of the evaluation participants suggested that it is not clear to the user why no further element is updated in the CFA graph while navigating through multiple rows in the error-path table. This led to the idea to mark the corresponding node label in addition to the node itself when navigating through multiple error path elements represented by a combined node in a CFA graph, as shown in Figure 19.

Reverse postorder id. The reverse postorder id available for CFA nodes was initially displayed as a second node label which matches the previous solution as can be seen in Figure 1. The evaluation results led to removing the reverse postorder id value from the node label and placing it in the tooltip displayed when the mouse cursor is positioned over the node, thereby reducing the labels in the CFA nodes.

6 Conclusion

During the course of this thesis a new approach for the generation of the CPAchecker report was introduced. The goal was to improve the previously existing solution in regards of performance, usability and available functionalities while keeping as much of the available implementation as possible in order to ensure fast and smooth transition. The preceding solution required additional software to be installed and an external script that was provided in the CPAchecker download to be executed by the users in order to generate the verification report. This was necessary so that the control-flow automaton and abstract reachability graphical representations can be created and included in the HTML report. The approach presented in this thesis does not require the installation or usage of any additional software besides CPAchecker. The verification report is generated automatically as part of the verification run. This reduces the required report-creation steps that need to be taken by the users thereby improving the overall user experience. The automatic report generation is achieved by providing the necessary data for CFA and ARG creation dynamically and in a structured way to the HTML report. The graphs are generated when the page is viewed using the third party JavaScript libraries D3 and Dagre-D3. In order to increase the performance and to avoid the well known "script is taking too long" webbrowser message when the report is opened, the solution presented in this thesis uses web workers as means to achieve multithreading in JavaScript. This provides the main script with the possibility to take care of smaller, inexpensive tasks and delegate costly computational tasks, like graph objects creation, to background-running operation-system level threads. The message exchange between the main script and the web workers is based on a well-defined protocol to ensure flawless communication among the scripts. In the previous solution the CPAchecker report could not always be created due to the external script not terminating while creating the CFA and ARG graphs. By dividing the work required to generate the report between multiple threads the solution presented in this thesis not only increases performance in regards of time needed to create the report, it also ensures the termination of the scripts. Furthermore, the new architectural and implementational approach solves the majority of the known issues connected with the previous version of the CPAchecker report generation. Additionally it keeps the interactive functionalities that were available in the earlier solution with slight modifications that aim to increase performance and provide better user experience. The new solution, presented in this thesis, provides additional report interaction capabilities to its users that include a pan event for the generated graphs and a special ARG graph, created if a counterexample is found by the CPAchecker verification, containing only error-path elements. In summary, the CPAchecker report-generation approach presented in this document does reach the goal of improving the previously available solution in regards of performance, usability and provided interaction capabilities, which was also reflected in the conducted evaluation.

6.1 Future Work

In this subsection we will focus on a couple of ideas that were born during the realization of the new CPAchecker report-generation approach and can be used to further improve the generated report or the report-generation process.

Dependency reduction. The report creation depends on multiple third party libraries that were either kept from the previous solution, because they are connected to some of the available functionalities in the report, or were introduced as aids to the new architectural and implementational approach. During the realization of the new solution the dependency

to the AngularJS library was heavily reduced and can, at future point in time, be decoupled from the CPAchecker report. The tasks which are currently handled by the library can be achieved by using one of the other third party JavaScript libraries which the report uses. For example the dynamic population of HTML elements based on the report data provided by CPAchecker can be achieved with D3 and the functionalities provided by the AngularJS controllers can be accomplished through the use of jQuery and Bootstrap or even native JavaScript. Reducing the dependencies will not only increase the maintainability of the report, it will also increase performance.

Automated tests. CPAchecker is a living project that is extended and adjusted on a daily basis. Providing automated tests for the content and functionalities included in the HTML report will not improve it in regards of usability, performance and available interaction functionalities but will ensure that it is not damaged by changes in CPAchecker. Having to manually verify that the CPAchecker report is not governed by changes in the software is a time consuming task due to the variety of available interaction functionalities. Automatically executed tests will ensure that if the report is negatively affected by changes in the source code, the developers will be made aware of such implications. There are multiple open source frameworks that provide the possibility to create automated tests for JavaScript and HTML. To name a few of the most popular ones, Mocha and Jasmine.

Statement of originality

I hereby confirm that I have written the accompanying Bachelor thesis by myself, without contributions from any sources other than those cited in the text and acknoledgements.
This applies also to all graphics, drawings and images included in the thesis.

Munich, 10/17/2017

.....
Deyan Ivanov

A Questionnaire

Interactive Visualizsation of Verification Results from CPAchecker with D3

Evaluation

1. General questions

1.1. Are you familiar with CPAchecker?

Yes ☐ No ☐

1.2. Have you used the previous version of the reports generated by CPAchecker?

Yes ☐ No ☐

1.3. How often do you use CPAchecker including the generated reports?

a. Daily
b. Weekly
c. Monthly
d. Never
e. _____

2. User Tasks – First task

Please execute the '**First task**' defined in the User Tasks document before you proceed with the questionnaire.

3. Report analysis and feedback

3.1. Was it clear where the generated report can be found?

Yes ☐ Not really ☐ Not at all ☐

3.2. Is the feedback during graph rendering sufficient and helpful to the user?

Yes ☐ Not really ☐ Not at all ☐

3.3. Are the "Help" buttons easy enough to find?

Yes ☐ Not really ☐ Not at all ☐

3.4. Would it be more helpful for the user if the "Help" boxes are opened automatically directly after graph rendering is complete?

Yes ☐ Not really ☐ [1] Not at all ☐

3.5. (Only if you said 'Yes' in the previous question) How would you expect the "Help" boxes to close?

3.6. Is the information provided by the "Help" buttons understandable and sufficient?

Yes ☐ Not really ☐ Not at all ☐

CFA Tab

3.7. Is the default selection - "all" - in Displayed CFA a good choice?

a. Yes

b. No, preferred default:_____

3.8. Is the default behavior on 'Mouse wheel' what a user might expect? Would you prefer a different behavior? (Please consider zoom, scroll and pan)

3.9. Should the label count inside a 'combined node' be reduced? If so, please suggest an appropriate number of label rows to display. Should the remaining label rows be displayed on user interaction, i.e. on double click, inside a popup window?

3.10. Does the second node label value inside the circle and diamond shaped nodes, the reverse post order id, carry a significant value for the user? If it is to be removed, would you like to have it in the tooltip box?

3.11. Is the information provided by the tooltip box understandable and helpful at all? If 'no', please suggest an alternative to include inside the tooltip.

a. Yes

b. No, _____

3.12. Please rate the 'Jump to Source' event.

Doesn't work at all ☐ ☐ ☐ ☐ ☐ Works perfectly

ARG Tab

3.13. Is the default behavior on 'Mouse wheel' what a user might expect? Would you prefer a different behavior? (Please consider zoom, scroll and pan)

3.14. Is the information provided by the tooltip box understandable and helpful at all? If 'no', please suggest an alternative to include inside the tooltip

a. Yes

b. No, _____

3.15. Please rate the Displayed ARG selection to show only error path.

Not helpful ☐ ☐ ☐ ☐ ☐ Very helpful

3.16. Please rate the 'Jump to CFA' event.

Doesn't work at all ☐ ☐ ☐ ☐ ☐ Works perfectly

Error-Path Window

3.17. Please rate the search functionality of the error-path-window

Works poorly ☐ ☐ ☐ ☐ ☐ Works great

3.18. Please rate the error-path walk along (as an average for all tabs) with 1 being the highest score

5 ☐ ☐ ☐ ☐ ☐ 1

Error Analysis

3.19. Were you able to easily locate the error using the generated report?

Yes ☐ Not really ☐ Not at all ☐

3.20. In which Source code line does the error occur? Which variable (and value) causes the error?

Line:_____ Variable:_____

4. User Tasks – Second task

Please execute the '**Second task**' defined in the User Tasks document before you proceed with the questionnaire.

5. Layout and overall user experience

5.1. Do you agree with the general placement and sizing of the 'error path' window and the 'tabs' window?

Yes ☐ No ☐, please use the last section to provide more feedback.

5.2. Do you agree with the overall used graph element shapes and colors?

Yes ☐ No ☐, please use the last section to provide more feedback.

6. Comparison to the previous solution

Please consider this section only if you are familiar with the previous report!

6.1. How would you rate the new solution in regards of performance?

No improvement at all ☐ ☐ ☐ ☐ ☐ Immensely improved

6.2. How would you rate the new solution in regards of usability?

No improvement at all ☐ ☐ ☐ ☐ ☐ Immensely improved

6.3. How would you rate the new solution in regards of functionality?

No improvement at all ☐ ☐ ☐ ☐ ☐ Immensely improved

Additional feedback

Use this section to add any additional comments, remarks, notes and criticism regarding the report page.

B User Tasks

User Tasks

1. First task

1.1 Preparation

Please review the Quick Reference documentation found here:

https://github.com/sosy-lab/cpachecker/blob/trunk/doc/Report.md

1.2 Obtain data

Download the file bubble_sort_false-unreach-call.i from the CPAchecker Github repository by using the link:

https://github.com/sosy-lab/sv-benchmarks/blob/master/c/loops/bubble_sort_false-unreach-call.i

1.3 Run the CPAchecker analysis

`scripts/cpa.sh -predicateAnalysis $path-to-file/bubble_sort_false-unreach-call.i`

If you are using Windows OS outside of a Cygwin environment please exchange **`scripts/cpa.sh`** with **`scripts/cpa.bat`**.
Make sure to enter the actual path to the downloaded file and to use the correct initialization script!

1.4 Open the generated report

- View the information provided in the "Help" buttons

- Make yourself familiar with the following functionalities

  - Change the displayed CFA function to "inspect"

    - Scroll along the graph (mouse wheel) and make yourself familiar with the layout

    - Check the "Mouse Wheel Zoom" box
  - Note the mouse wheel behavior while having the pointer over a graph
  - Note the graph behavior when you *drag and drop* the graph, also known as *pan*– make sure the mouse pointer is **not** placed on a graph element
    - Hover over some node and edge elements and make yourself familiar with the available information provided in the tooltip

- Double click on any CFA edge to jump to the relating Source code line. Switch back to the CFA tab afterwards

- Change the displayed CFA function back to "all"

- Switch to the ARG tab

  - Scroll along the graph (mouse wheel) and make yourself familiar with the layout

  - Check the "Mouse Wheel Zoom" box and verify the same behavior as for CFA graphs

  - Hover over some node and edge elements and make yourself familiar with the available information provided in the tooltip

  - Use Displayed ARG to show only the error path ARG graph

  - Double click an ARG node to jump to the relating CFA location

- Use the search functionality in the Error Path Window

  - Search for the value 'tmp' without selecting the exact matches check box

  - Search for the value 'tmp' after selecting the exact matches check box

  - Select one of the CFA, ARG or Source tabs and let yourself walk along the error path

  - Select any other tab and let yourself walk along the error path again

Now that you have made yourself aware of the available functionalities and their behavior please analyze the program. Find which variable (and its value) cause the error. Please note yourself the source code line in which the error occurs as well as the variable and its value that cause the error.

Please continue with the questionnaire.

2. Second task

Download the bist_cell_true-unreach-call_false-termination.cil.c file from:
https://github.com/sosy-lab/sv-benchmarks/blob/master/c/systemc/bist_cell_true-unreach-call_false-termination.cil.c
and run CPAchecker with the following command line.

`scripts/cpa.sh -predicateAnalysis $path-to-file/bist_cell_true-unreach-call_false-termination.cil.c`

Switch to the ARG Tab and scroll down until you see an edge pointing to an labelless node. Hover over the node and inspect the tooltip. Use the described event to jump to the actual target node contained in the other part of the graph. Double-click on the 'split edge' pointing to the marked node to jump back to the starting point

Please continue with the questionnaire.

References

[1] Wilhelm Barth, Michael Juenger, and Petra Mutzel. *Simple and Efficient Bilayer Cross Counting*, pages 130–141. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.

[2] Dirk Beyer and M. Erkan Keremoglu. *CPAchecker: A Tool for Configurable Software Verification*, pages 184–190. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[3] Ulrik Brandes and Boris Alexander Koepf. Fast and simple horizontal coordinate assignment. In Petra Mutzel, Michael Jnger, and Sebastian Leipert, editors, *Graph Drawing*, number 2265 in Lecture Notes in Computer Science, pages 31–44, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[4] E. R. Gansner, E. Koutsofios, S. C. North, and K.-P. Vo. A technique for drawing directed graphs. *IEEE Trans. Softw. Eng.*, 19(3):214–230, March 1993.

[5] Michael Juenger and Petra Mutzel. 2-layer straightline crossing minimization: Performance of exact and heuristic algorithms. *Journal of Graph Algorithms and Applications*, 1:Paper 1, 25 p.–Paper 1, 25 p., 1997.

[6] Magdalena Murr. Towards understandable cpachecker counterexamples. unpublished thesis, 2016.