UAV: Warnings from Multiple Automated Static Analysis Tools at a Glance

Tim Buckers, Clinton Cao, Michiel Doesburg, Boning Gong, Sunwei Wang, Moritz Beller, Andy Zaidman
Delft University of Technology, The Netherlands
{t.b.buckers,c.s.cao,m.s.doesburg,b.gong,s.wang-11}@student.tudelft.nl {m.m.beller,a.e.zaidman}@tudelft.nl

***Abstract*—Automated Static Analysis Tools (ASATs) are an integral part of today's software quality assurance practices. At present, a plethora of ASATs exist, each with different strengths. However, there is little guidance for developers on which of these ASATs to choose and combine for a project. As a result, many projects still only employ one ASAT with practically no customization. With UAV, the Unified ASAT Visualizer, we created an intuitive visualization that enables developers, researchers, and tool creators to compare the complementary strengths and overlaps of different Java ASATs. UAV's enriched treemap and source code views provide its users with a seamless exploration of the warning distribution from a high-level overview down to the source code. We have evaluated our UAV prototype in a user study with ten second-year Computer Science (CS) students, a visualization expert and tested it on large Java repositories with several thousands of PMD, FindBugs, and Checkstyle warnings. Project Website: https://clintoncao.github.io/uav/**

I. Introduction

Automated Static Analysis Tools (ASATs) analyze source or binary code without observing its run time behavior [1]. ASATs have become an integral part of today's software quality assurance practices, reflected by the increased uptake of new ASATs such as Google's Error Prone[1] or Facebook's Infer.[2] In addition to the well-known standalone ASATs like FindBugs, Checkstyle, or PMD [1], recently, new cloud services like CodeClimate[3] have emerged. They try to provide a better integration of ASATs into the development process. As a result, at present, a plethora of ASATs exist, each with different strengths and many different warning typologies.

However, developers have little guidance which ASATs to choose and combine for a project. Developers lack a tool that allows them to explore *which* warnings a *certain* ASAT emits *where* in the project, and whether there are *overlaps* with existing ASATs. Currently, developers and researchers can only run ASATs individually and then compare their output, which is both tedious and leaves many features to be desired. As a result, many projects still only employ one ASAT with practically no customization and never explore the possibility of combining multiple ASATs to their benefit [1].

To address this issue, we have created UAV, the Unified ASAT visualizer. UAV can run multiple ASATs and facilitates comparing them by unifying their different warning typologies and representing all warnings in one interactive treemap visualization. For researchers, UAV offers a flexible means to analyze the different types of warnings generated by multiple ASATs. For software developers, our tool gives insight into the warning distribution in their Java projects. After locating a specific class full of warnings, developers can seamlessly navigate to the source code view where the relevant warnings will be highlighted. UAV can also support ASAT tool creators themselves by helping them sharpen the focus of their tools: They can compare the warning types their tool detects to its competition and thus differentiate themselves better. In the remainder of this paper, we describe UAV from an end-user as well as a technical perspective.

[1]http://errorprone.info/
[2]http://fbinfer.com/
[3]https://codeclimate.com/

II. User Story

Bob and his team of software engineers at XYZ Inc. are developing a revolutionary new search website. They decided to use multiple ASATs to ensure a basic level of code quality, including Checkstyle and FindBugs. After a few weeks of development, Bob checks all warnings. To his surprise, the ASATs report a list of over a thousand warnings on the relatively new project. Bob wants to address the warnings in an efficient manner, but has no idea where to start. He is discouraged by the fact that he cannot get an overview of how the warnings are distributed across the system's components. For example, warnings related to the search subsystem would take precedence over warnings in the user interface (UI) components of the new search website. Bob knows that working through the lengthy list of warnings one at a time will be extremely time-consuming, but sees no other option. Working through the list Bob repeatedly notices overlaps between warnings from different ASATs, albeit under slightly different names. For example, for the method `AdvanceState`, Checkstyle and FindBugs emit the overlapping warnings `MethodName` and `NM_METHOD_NAMING_CONVENTION`. Bob realizes that he is losing significant time on similar issues. Moreover, he has no way to exclude warnings which are irrelevant to his team. Bob wonders: Isn't there a tool which provides me with …

- an overview of where in the system warnings are concentrated?
- an overview of warnings which have the highest priority?
- a way to filter irrelevant warnings?
- a way to filter overlaps in the warnings from multiple ASATs?

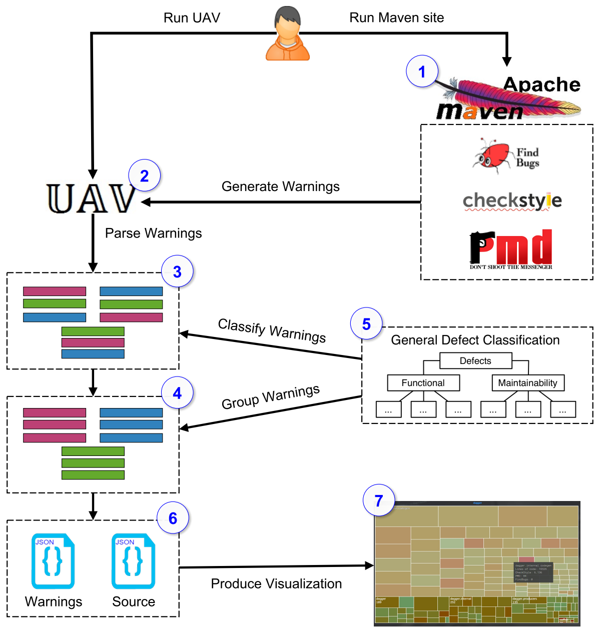


Fig. 1. Workflow of UAV.

III. Related Work

In this section, we give an overview of literature and tools that are related to UAV.

A. Literature

In recent work on the state of the art of ASATs [1], we introduced the General Defect Classification (GDC). The GDC is a topology that allows the categorization of warnings from different ASATs into a set of mutually shared categories. It forms the basis of UAV's visualization.

Many ASATs differ in the type of defects they detect. However, even when tools focus on uncovering the same category of defect type, the variance in the concrete warnings they emit and their naming is still very large [1]. This indicates that using several ASATs has benefits over using a single ASAT. Using multiple ASATs can be time consuming, however, arbitraging a single warning can take up to eight minutes on average [2]. Moreover, ASATs generate about 40 warnings per 1,000 lines of code [3]. With UAV, developers and researchers can visually assess this rich and plentiful torrent of warnings for the potential benefits of combining multiple ASATs. It enables developers to make an informed decision on whether the added findings and their type justify the inclusion of another ASAT into their tool chain. Researchers have long performed comparative studies with multiple ASATs and other quality assurance techniques like code review, for example Wagner et al. in 2005 [4] and Panichella et al. in 2015 [5]. However, they lacked a tool that allows them to visually compare the location and defect types of different ASATs on concrete real-world projects. UAV closes this gap.

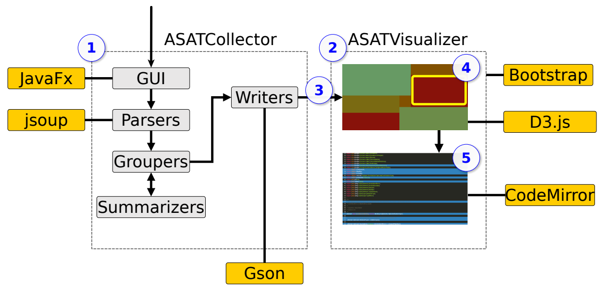


Fig. 2. Architecture of UAV.

B. User Workflow

UAV offers a visual way of exploring which packages or classes are particularly affected by ASAT warnings. By contrast, existing research has tackled the problem of how to deal with a flood of warnings mainly by prioritizing them. Muske and Serebrenik give a comprehensive overview of the approaches that have been suggested so far [6].

To visualize data in a structured way, UAV uses treemaps on its package and class level views and an enriched source code view on individual files. Treemaps are a space-filling visualization method that can display large hierarchical collections of quantitative data intuitively [7]. This makes it ideally suited to present the nested structure of a typical Java project. UAV uses a modified treemap view to provide an intuitive high-level visualization of which warnings lie where in a project and a seamless switch to a source-level view to track warnings down to individual source code lines.

C. Tools

Apart from the plethora of individual ASATs available today, tools such as Teamscale [8], SonarQube [9] and CoverityScan [10] can collect and display the warnings of multiple ASATs, the first step of UAV. UAV goes further in that it also categorizes the warnings from the multiple tools into one mutual topology, GDC, and visualizes them. Alternatively, UAV displays the ASAT warnings originate from, down to the source code level. Existing tools lack these two capabilities.

IV. Implementation

In this section, we first give an overview of the workflow for a user, then describe UAV's architecture and inner workings, and conclude with a series of technical challenges.

A. Workflow

Figure 1 depicts the typical workflow of UAV. It begins with the user running `maven site` to produce the warning files of the ASATs ①. The user then indicates, in UAV's UI ②, the source folder of the project to analyze. UAV gathers context data on the project and parses the generated ASAT warnings ③. Subsequently, it classifies and groups warnings ④ by applying the GDC ⑤ on them. Next, it writes out the result files for the visualizer ⑥. Finally, UAV opens the user's web browser and runs the visualizer ⑦.

B. Architecture

Figure 2 depicts the two components UAV comprises. The ASATCollector ① gathers and interprets the output generated by running the supported ASATs via Maven. Because of its static and computation-intense nature, we have implemented the ASATCollector in Java 1.8. The ASATVisualizer ②, allows a user-interactive exploration of these warnings transferred from the ASATCollector ③. To emphasize platform independence, speed, and user interaction capabilities, we have implemented the visualizer in JavaScript to run in the user's browser.

The ASATCollector first finds all warnings, along with their specific location in the project, and groups them together. Second, it determines the structure of a project. When one runs UAV, the ASATCollector will open up a JavaFX UI where the user can select the source folder of the project. Once selected, the parsers of the ASATCollector read the warning files generated by `maven site` for Checkstyle, FindBugs and PMD. We use jsoup to parse GDC's ASAT mapping, specifying which ASAT warning to map to which common GDC category. The groupers summarize these warnings according to the read-in GDC. Simultaneously, the ASATCollector gathers information on the structure of the project by looking up all classes within each package, the path to each Java class file and the number of lines of code for each class and package. The last step is to write the collected warnings and data to a JavaScript file where it is stored in JSON format and transferred to the ASATVisualizer. The Gson library is used for the creation of JSON objects.

After it creates the output file, UAV opens the user's default browser and shows the visualization. Moreover, users can share its light-weight output file and without having to distribute the visualization code. This also means that multiple users can analyze the produced warnings of the project without having to run the independent ASATs multiple times. The visualization itself is a ready-made template based on the Bootstrap framework using HTML, CSS, and JavaScript. It only requires a JSON output file from the ASATCollector to display its information. In the ASATVisualizer, the treemap in the center of the visualization is implemented using D3.js, a popular JavaScript library for manipulating documents based on data. We have chosen D3.js because of its interactive features and freedom of customization. This enabled us to implement the different filter options of the treemap in pure JavaScript. If the user clicks on a class in the treemap, UAV will seamlessly swap the treemap with the source code viewer. The source code viewer is built using the JavaScript library CodeMirror; we modified the syntax highlighting to show the warnings at the source code level with color-coding.

C. ASATVisualizer User Interface

In this section, we describe the two main UI components of UAV: Its treemap high-level package view depicted in Figure 3 and its source code-level view in Figure 4.

UAV's visualization provides users with a large treemap showing the structure of the project (① in Figure 3). The treemap can be navigated through by clicking on the desired block. Currently, the package 'dagger.internal.codegen' is highlighted ②. Next to the mouse cursor, UAV displays a pop-up with descriptive statistics about the highlighted package, such as its number of warnings per ASAT ③. The user could click on this package to zoom in on it. In the menu on the left ④, users can select which ASATs to include in the visualization. They can adjust which metric the color of the classes are based on:

- 'Normal' shows the amount of warnings relative to other classes.
- 'ASAT' shows the distribution of which ASAT the warnings originate from.
- 'Category' shows the distribution of warnings according to which of the GDC categories (functional defects, maintainability defects, or other) they belong to.

When in 'Normal' color scale, users can also choose to base the intensity of the colors on the relative amount of warnings in each class or on an absolute scale (where pure green means no warnings and pure red means one warning per line). In the GDC panel on the right-hand side ⑤, the user can see the warning categories and toggle them on or off.

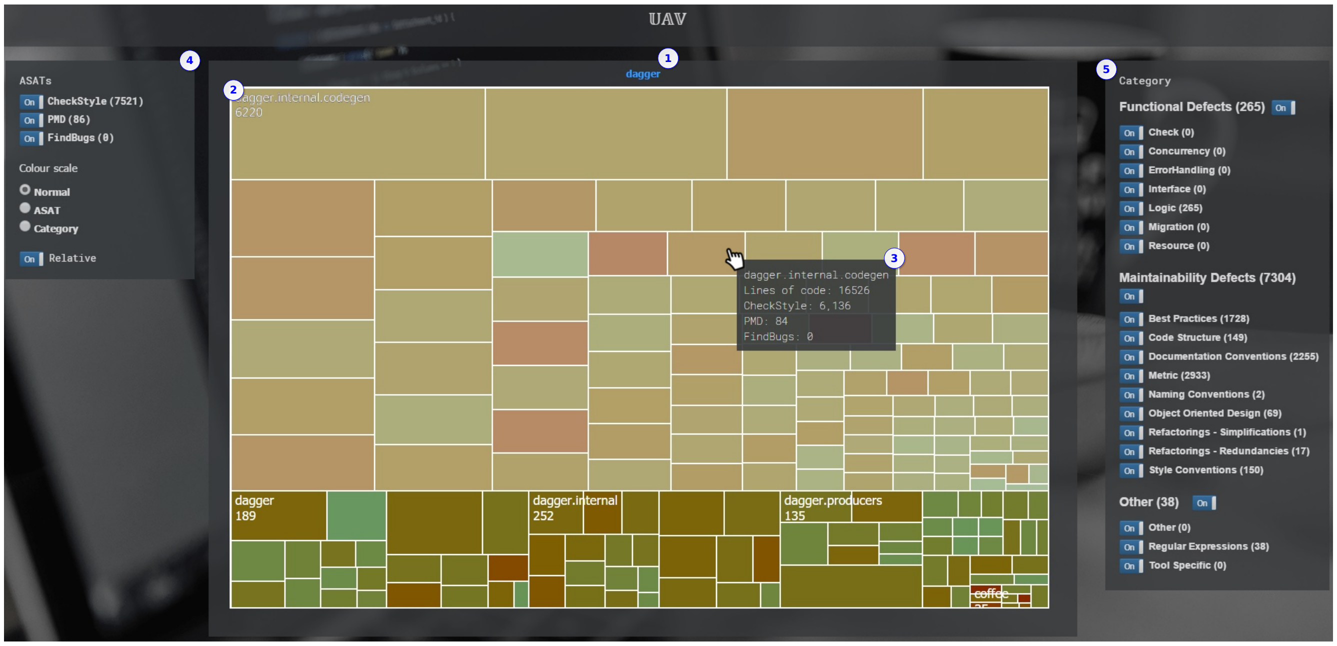
The user can navigate down from package level into class level view, and finally view a single class on code level, shown in Figure 4. UAV color-codes each line with a warning, see line 16. According to the setting of ④ in Figure 3, the color can indicate which ASAT the warnings originate from or which category they belong to. In lines with multiple warnings, colors alternate, see line 5. It contains a warning about code structure, namely the import 'java.io.IOException' is not used. Both PMD and Checkstyle have reported this warning.

D. Challenges

We have encountered three major challenges during the development of UAV. Our first challenge was to find a way for UAV to run the ASATs. The initial solution was to use processbuilder from Java; it is possible to run commands via processbuilder to execute the ASATs. However, this solution required an executable of each ASAT, which restricts our users to one version of an ASAT and makes it difficult to update. Our alternate solution is to use Maven to produce the output files of the ASATs. For this solution there is no need to package third-party executables of ASATs together with UAV.

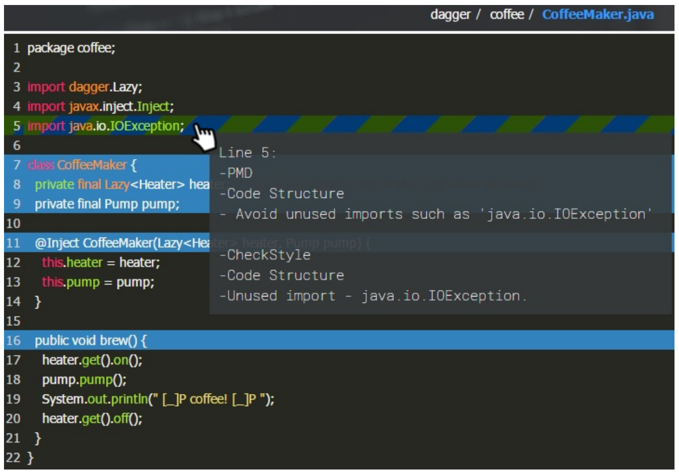
The solution for the first challenge, however, is a cause of the remaining two challenges: we had to find a way for UAV to run Maven and to gather all the output files of the ASATs. As Maven can be run as a standalone application, installed in the system, or incorporated in an integrated development environment (IDE), it is difficult to determine and support all three possible installation scenarios through UAV. Instead, we have therefore decided to let the user run Maven on their project before they use UAV.

Finding the warning files of ASATs is a straight-forward task as long as a project only contains one Maven configuration file. However, in many larger projects, each package has its own pom configuration file, which produces its own ASAT



```
dagger.internal.codegen
Lines of code: 16526
CheckStyle: 6,136
PMD: 84
FindBugs: 0
```

Fig. 3. High-level package view of UAV on the Dagger project.



```
package coffee;

import dagger.Lazy;
import javax.inject.Inject;
import java.io.IOException;

class CoffeeMaker {
  private final Lazy<Heater> hea
  private final Pump pump;

  @Inject CoffeeMaker(Lazy<Hea
    this.heater = heater;
    this.pump = pump;
  }

  public void brew() {
    heater.get().on();
    pump.pump();
    System.out.println(" [_]P coffee! [_]P ");
    heater.get().off();
  }
}
```

```
Line 5:
-PMD
-Code Structure
- Avoid unused imports such as 'java.io.IOException'
-CheckStyle
-Code Structure
-Unused import - java.io.IOException.
```

Fig. 4. Code-level view of UAV with Checkstyle and PMD warnings.

outputs. Hence, before UAV can work on these, it must unify them into a single warning file per ASAT.

V. Evaluation

In this section we report on initial evaluations of UAV on three real-world systems and a usability study with ten CS students.

A. Project Case Study

To evaluate whether UAV can be used on larger real-world projects, we tested it on two popular Java projects from GitHub, google/dagger (5,292 stars)[4] and apache/curator (486 stars),[5] and on itself (the Java part of UAV, ASATCollector

[4]https://github.com/google/dagger
[5]https://github.com/apache/curator

TABLE I
Descriptive statistics of UAV on three example projects.

Name	#LOC	#Checkstyle warnings	#PMD warnings	#FindBugs warnings	Run time
google/dagger	59,864	7,521	86	0	73s
Netflix/curator	122,094	16,691	53	0	156s
UAV	4,796	5	20	14	1s

in Figure 2). For each project, we ran the tool ten times and calculated the average run time. We measured the run time from when the tool starts to gather all information of the user's project to the point where the analyzer writes the output files for the ASATVisualizer. Table I shows descriptive statistics and the run time of our tool on each project.

An interesting result from comparing the tree projects was that the amount of warnings per tool depended on the project, their specific ASAT configuration. For example if many Checkstyle rules are removed or FindBugs is set to a lower rigidity, then the amount of warnings is visibly reduced in UAV. We could compare and observe the effect of modifying the project's configurations via UAV's "absolute" color scheme (see Section IV-C). Thus, UAV also provides insights in the development stage of software.

B. User Study

We invited ten second year computer science students and later a visualization expert (both with no prior knowledge of UAV) to participate in our usability testing. We placed them in front of a computer with UAV, accompanied by a list of questions, and a short explanation of the purpose of the tool. The testers could interact with the tool while answering questions related to its use. Questions like "Which package has the most warnings?" and "How many warnings in the project are about Code Structure?" helped us assess how intuitive to use UAV was by measuring how many students delivered a correct answer. The last question was an open question where the testers were asked for further feedback. We replicate the list of all questions and the in-depth results of the usability evaluation in an online appendix [11].

Our results indicate that most testers understood the goal of the tool. At least 70% of respondents answered each question correctly. Based on incorrect answers and the feedback given in the evaluation, we could improve the tool in several ways. One such improvement is the backwards navigation bar. One of the testers said: "The back button on the top looks like you can go back to a specific folder instead of the previous folder." This feature was initially designed to allow users to go one level up in the visualization of their project. After discussions within the team, we replaced the navigation feature with the current path to the file which the user is viewing. Moreover, we made each component of the path itself clickable. We could implement several more improvements in the UI and UAV's usability. Later feedback from the visualization expert showed us that this made the navigation of the tool more intuitive [11].

VI. Future Work

In this section, we will describe possible improvements and extensions of UAV for future work.

Due to compatibility issues with the treemap visualization and the gradient color representation of D3.js, Chrome and Safari are the only supported browsers at this time. We plan to resolve the cosmetic problems with Firefox.

UAV's visualization of nested packages could be improved. It currently does not show the nested relationship of sub-packages, but rather includes them on the top-level of the treemap. Implementing this feature would allow UAV to handle more hierarchically complex projects.

The current UAV prototype supports three Java ASATs. A natural improvement would be adding more ASATs to broaden the selection of tools that can be compared by including tools such as Google's Error Prone. The ASATCollector facilitates adding new ASATs thanks to its modular structure. We would only need to change the UI of the ASATVisualizer to handle the visualization of additional tools. Supporting more tools and programming languages would also lift UAV's status of a prototype.

A promising avenue of future work would be the integration of UAV with GitHub and Travis CI, a cloud service that automatically builds GitHub projects. Similar to CodeClimate, a new commit on GitHub could trigger the execution of Maven on Travis CI, store the ASAT warnings as build artifacts, and UAV in the cloud would collect these artifacts and generate a JSON file for the visualization. The existing visualization implementation of UAV lends itself toward such hosting in the cloud, since it is based on a web-stack and would only require the relatively light-weight visualization file.

VII. Conclusion

In this paper, we present UAV, a tool that provides an intuitive way to compare multiple ASATs. UAV makes the following key contributions:

- A novel structured, interactive visualization that allows for comparison between multiple ASATs.
- Configuration options to switch the visualization between the amount of warnings per ASAT, package, class and GDC defect type.
- A basic framework that can be expanded to include more ASATs and comparison methods as well as additional features.
- A clear overview of warnings from different ASATs in large real-world software projects.

In our first evaluation, our UAV prototype has demonstrated its capability of visualizing warnings by clearly representing multiple Java projects of different project sizes and ASAT warning densities. Users of our tool have a more coherent view of the types and locations of warnings as indicated by different ASATs. Our vision is that one day, anyone who uses code analysis can input their preferences, and UAV will combine different ASATs to output a result that best suits their needs.

Acknowledgments

We thank Bastiaan Reijm for the help that he provided throughout the development of UAV, Fabian Beck for useful suggestions on the first release candidate, and all students who participated in our usability evaluation.

References

[1] M. Beller, R. Bholanath, S. McIntosh, and A. Zaidman, "Analyzing the state of static analysis: A large-scale evaluation in open source software," in *Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2016, pp. 470–481.

[2] J. Ruthruff, J. Penix, D. Morgenthaler, S. Elbaum, and G. Rothermel, "Predicting accurate and actionable static analysis warnings: an experimental approach," in *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 2008, pp. 341–350.

[3] S. Heckman and L. Williams, "On establishing a benchmark for evaluating static analysis alert prioritization and classification techniques," in *Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM)*. ACM, 2008, pp. 41–50.

[4] S. Wagner, J. Jürjens, C. Koller, and P. Trischberger, *Comparing Bug Finding Tools with Reviews and Tests*, ser. LNCS. Springer, 2005, vol. 3502, pp. 40–55.

[5] S. Panichella, V. Arnaoudova, M. Di Penta, and G. Antoniol, "Would static analysis tools help developers with code reviews?" in *Proc. International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2015, pp. 161–170.

[6] T. Muske and A. Serebrenik, "Survey of approaches for handling static analysis alarms," in *Proc. International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2016, pp. 157–166.

[7] B. Johnson and B. Shneiderman, "Tree-maps: A space-filling approach to the visualization of hierarchical information structures," in *Proc. of the 2nd Conference on Visualization (VIS)*. IEEE, 1991, pp. 284–291.

[8] L. Heinemann, B. Hummel, and D. Steidl, "Teamscale: Software quality control in real-time," in *Companion Proceedings of the Int'l Conference on Software Engineering (ICSE)*. ACM, 2014, pp. 592–595.

[9] G. Campbell and P. P. Papapetrou, *SonarQube in Action*. Manning Publications Co., 2013.

[10] "Coverity Scan Static Analysis," https://web.archive.org/web/20161124164054/https://scan.coverity.com/.

[11] T. Buckers, C. Cao, M. Doesburg, B. Gong, S. Wang, M. Beller, and A. Zaidman, "Online Appendix for UAV: Warnings From Multiple Automated Static Tools At A Glance," https://figshare.com/s/05658ac8ff03d57a8d60.