SAW-BOT: Proposing Fixes for Static Analysis Warnings with GitHub Suggestions

Dragos Serban
Eindhoven University of Technology
The Netherlands
d.m.serban@student.tue.nl

Bart Golsteijn, Ralph Holdorp
Philips Research
The Netherlands
{bart.golsteijn,ralph.holdorp}@philips.com

Alexander Serebrenik
Eindhoven University of Technology
The Netherlands
a.serebrenik@tue.nl

***Abstract*—In this experience report we present SAW-BOT, a bot proposing fixes for static analysis warnings. The bot has been evaluated with five professional software developers by means of a Wizard of Oz experiment, semi-structured interviews and the mTAM questionnaire. We have observed that developers prefer GitHub suggestions to two baseline operation modes. Our study indicates that GitHub suggestions are a viable mechanism for implementing bots proposing fixes for static analysis warnings.**

***Index Terms*—static analysis, bot, GitHub suggestions**

I. Introduction

Static analysis tools can be used to detect potential code smells, bugs, styling violations or even security issues solely by analyzing the source code [1]. Although static analysis tools are widely known for their benefits, these tools are still not widely adopted, due to such limitations as lack of automated fixes, displaying too many alarms at once and the high number of false positives [2]. In fact, Vassalo et al. [1] have shown that one of the most important factors considered by developers when deciding which static analysis warnings to fix is the estimated fixing time.

Several solutions have been proposed for automatic fixing of static analysis warnings. SpongeBugs [3] provides automatic detection and fixing for 11 static analysis violations that are detected by both SonarQube and FindBugs. SpongeBugs generates fixes for the entire project; the fixes are directly introduced in the source code. Tricorder [4] provides primarily indications of violations on a project-level, but for some violations quick fixes could be proposed as well, at code review time. Sonarqube-repair [5] provides automatic fixes for certain SonarQube warnings via pull requests. Finally, AVATAR [6] and Phoenix [7] generate fixes based on patch mining and learning approaches. Similar to SpongeBugs, these tools do not propose the fixes as suggestions, but they perform the changes directly into the code base.

While directly changing the code base reduces the effort developers need to spend addressing static analysis warning, it can only be successful if developers trust an automatic tool. Although in the long run such a trust relation is possible, adoption of such a "direct fix" approach might be expected to be slowed down by lack of trust. Hence, we believe that an automatic tool should suggest changes to the developers that can accept or reject them. Moreover, such an automatic tool should not disrupt the developers' workflow by presenting with a "wall of warnings" (cf. [2]) and provide feedback as soon as possible, i.e., when a pull request is submitted.

In this paper we present SAW-BOT (Static Analysis Warnings Bot). SAW-BOT is based on SonarQube, a well-known static analysis tool. It invokes SonarQube analysis for a pull request submitted by a developer, generates suggestions for fixes to the SonarQube warnings, and integrates these suggestions directly in the pull request discussion.[1]

Remainder of the paper is structured as follows. We discuss the design of SAW-BOT in Section II. To understand how developers interact with SAW-BOT we design the evaluation study in Section III and discuss its results in Section IV. In Section V we review the threats to validity of our study and conclude in Section VI.

II. Design of SAW-BOT

SAW-BOT operates on a Github repository. It is implemented as a Github App[2] using the Probot[3] framework. Whenever a new pull request is opened, SAW-BOT invokes SonarQube to perform static analysis (Section II-A); generates a fix to the static analysis warnings (Section II-B); and reports the suggested fixes to the developer (Section II-C). As such SAW-BOT belongs to a large group of code-improving bots such as Refactoring-Bot [8] and Repairnator [9].

A. Static analysis

This project has been conducted in collaboration with Philips Research, which determined our decision to target Java applications. SonarQube[4] is a light-weight [10] rule-based static code analysis tool; for Java it has 612 rules [11]. We have selected the rules for which syntactic fixes can be automatically generated: (i) string literals should be on the left side when checking for equality; (ii) `Collection.isEmpty()` should be used to test for emptiness; (iii) code should not be commented out; (iv) unused local variables should be removed; and (v) "public static" fields should be constant. We include (i), (iv) and (v) as violations of these rules have

[1] https://docs.github.com/en/free-pro-team@latest/github/collaborating-with-issues-and-pull-requests/incorporating-feedback-in-your-pull-request

[2] https://developer.github.com/apps/about-apps/

[3] https://probot.github.io/

[4] https://www.sonarqube.org/

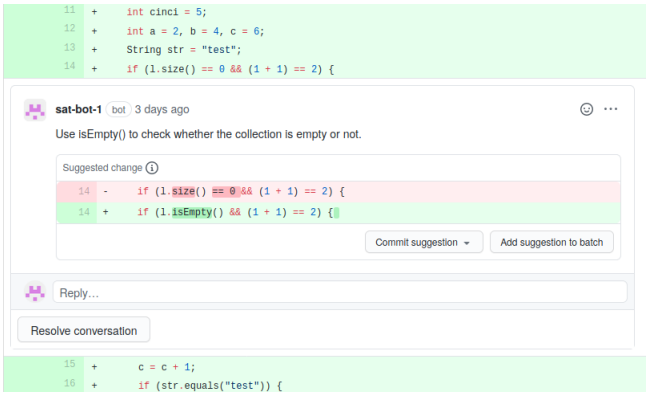

Fig. 1. SAW-BOT in the GitHub Suggestions mode

been shown to be likely to induce bugs [12]; (ii) and (iii) since violations of these rules have been frequently observed on the investigated Philips Research code repositories.

B. Generating fixes

The fix generation algorithm has been implemented using JavaParser,[5] a library that provides methods for the analysis and transformation of abstract syntax trees. We chose JavaParser since it can maintain the formatting and comments of the original source code after code transformations are applied.

C. Operation modes

As explained in the Introduction, the intended mode of operation of SAW-BOT involves analyzing the pull request submitted by a developer and integrating the proposed fixes in the code review, as illustrated in Figure 1. Developers can commit each suggestion individually by pressing "Commit suggestion", accept multiple suggestions as a single commit by clicking "Add suggestion to bulk" or "Resolve conversation" if they disagree with one or more suggestions. We call this mode the *GitHub Suggestions* mode.

To understand relative strengths and weaknesses of the GitHub Suggestions mode, we have also implemented two baseline operation modes: *Pull Request* mode and *Legacy* mode. Similarly to the GitHub Suggestions mode, in the Pull Request mode SAW-BOT proposes fixes only for warnings introduced in the analyzed pull request. As opposed to the GitHub Suggestions mode, in the Pull Request mode the fixes are proposed to the developers via a pull request to the branch created by the developer. Each individual fix is proposed as a separate commit. In the Legacy mode SAW-BOT is invoked at a predefined time. After performing the analysis, the bot generates fixes for all warnings that can be solved independently from when they have been introduced. Similarly to the Pull Request mode, the fixes are proposed to the developers via a pull request.

[5]https://javaparser.org/

III. Evaluation

To understand the benefits and drawbacks of the proposed approach we conducted an evaluation study with developers from Philips Research.

A. Design of the study

The evaluation study consisted of a Wizard of Oz experiment, augmented with a semi-structured interview. The Wizard of Oz experiments [13], [14] are widely used to evaluate systems before they are built. The method involves the intentional deception of the participant regarding the system that is being evaluated. In these experiments the participant is asked to perform a number of activities with a system. However, this system is in fact operated by a human, called the Wizard. The participant is not informed about this fact, in order to preserve the authentic behavior towards the system.

We opted for the Wizard of Oz experiment since it allowed us to evaluate a partially implemented SAW-BOT, and determine whether the GitHub Suggestion mode is preferred by developers compared to the baseline modes. The Wizard of Oz experiment has been designed according to the iterative methodology of Fraser et al. [15]. The experiment session was designed to take place on Microsoft Teams.

The Wizard of Oz experiment session started with the first author explaining the main features of SAW-BOT, as well as the three operation modes that would be evaluated. Then the participant is introduced to the GitHub repository[6] and given five minutes to familiarize themselves with the Java project and with the issues highlighted by the SonarQube instance connected to the repository.

Next the experimenter introduced the *Legacy mode*, i.e., informed the participant that SAW-BOT has been scheduled to open a pull request at a specific time. Instead the pull request was opened manually by the Wizard at the specified time. The Wizard operated using a specially created GitHub account: the name of this account was chosen to suggest that the account is used by a bot. The commits containing the fixes and the branch containing these commits were prepared before the start of the experiment on the local machine of the Wizard. In order to avoid compromising the illusion of realism, the authoring and commit date for each of these commits were tweaked, such that they all looked like they have been created at the specified time. Furthermore, one of the proposed fixes was intentionally wrong. After the Wizard pushed the commits containing the fixes and opened the pull request, the participant was asked to review the pull request. Special attention was paid to whether the participant used the revert command to remove the wrong fix. If the participant used this command, the Wizard manually submitted a new pull request that excluded the wrong fix.

Then the experiment moved to the *Pull Request* mode. The participant was requested to complete a simple coding task, which contained very specific implementation requirements, such as using certain methods. These requirements have been designed such that the produced code would trigger

[6]https://github.com/drgs/spring-boot-library-app-1

SonarQube warnings. When the coding task was finished, the participant was asked to create a new branch, push the created code to the remote GitHub repository and open a pull request to the main branch. After opening the pull request, the SAW-BOT invoked SonarQube and opened the pull request with the corresponding fixes. Since this operation mode was implemented in SAW-BOT at the time of the experiment, no actions were required from the Wizard.

Finally, while the SonarQube analysis was performed as part of the evaluation of the *Pull Request* mode, the Wizard manually submitted fixes using GitHub suggestions. After the bot successfully opened the pull request, the participant was asked, again, to review the new pull request. After the review was finished, the participant was informed that the evaluation of the *GitHub Suggestions* mode would start and they would be asked to review the suggestions proposed on the original pull request that the participant opened. Similarly to the *Legacy* mode, the Wizard intentionally proposed a faulty fix.

After the Wizard of Oz experiment, the first author conducted a semi-structured interview. The interview aimed at clarifying actions that the participant took during the Wizard of Oz experiment, and gathering additional insights regarding the overall user experience and the strengths/weaknesses of the proposed bot design. We used open coding and card sorting [16] to analyze the answers. To reduce the subjectivity of these methods, two persons have been involved.

Next we asked the participants to score SAW-BOT based on the modified Technology Acceptance Model (mTAM) [17] to evaluate the perceived usability of the proposed bot design. mTAM consists of a set of 12 statements on a 7-point scale, ranging from extreme disagreement to extreme agreement. This model quantifies usability based on two variables: perceived usefulness and perceived ease of use. The interpretation of perceived usability helps in understanding the acceptability of SAW-BOT in an industrial context. We opt for mTAM rather than the original Technology Acceptance Model (TAM) [18] since mTAM focuses on assessing the current experience with the system, rather than the expected experience.

At the end of the evaluation session, the participants were debriefed about the deception and the nature of the experiment, as recommended by Dahlback et al. [14]. In addition, the participants were given the opportunity to have all the collected data erased if they wished to do so.

B. Evaluating the study design

In order to test the study design, as recommended by Fraser et al. [15], we conducted five pilot sessions. As the result of the pilot studies we made several improvements to the initial study protocol, e.g., removing repeated questions and clarifying the coding task. The final study design has been approved by both the Ethical Review Board of Eindhoven University of Technology (ERB2020MCS12) and the Philips Internal Committee for Biomedical Experiments.

IV. Results

We have conducted the evaluation study with five professional developers from Philips Research: the number of participants is in line with previous studies [19] (albeit larger Wizard of Oz experiments have also been reported [20]). We used convenience sampling to recruit the participants.

A. Wizard of Oz

For the *Legacy* mode, the developers reviewed the proposed changes either per commit or directly in the "Files changed" view, which directly highlights all the changes that the bot proposed. None of the participants read the pull request description and none realized that the bot had a revert command, which allowed them to revert specific commits directly in the pull request. In addition, only one participant managed to identify the wrong fix in this operation mode, while others immediately merged the pull request after having a quick look at the proposed changes. Some participants claimed that the bot is not transparent about why a certain change has been made, even though the description of the fixed issues has been added in the commit message containing the fix. All participants ended up approving the pull request.

While the *Legacy* mode was considered acceptable, it did not encourage developers to review the changes carefully, which might lead to the introduction of bugs. Moreover, the bot should support displaying a comment for each of the fixed warnings, such that transparency is increased.

For the *Pull Request* mode, most developers started reviewing the changes directly in the "Files changed" view. Some developers read the description of the pull request and realized that a revert command existed. This is due most likely to the fact that the "Conversation" view of the pull request included a small number of commits (2 vs. 11 in the Legacy mode), which allowed developers to focus closer on what was displayed on the screen. All participants decided to approve and merge the pull request by squashing. Two participants mentioned the transparency issue again. Developers also recommend highlighting the warnings issued by SonarQube, but which could not be fixed by the bot.

The *GitHub Suggestions* was preferred by 4 out of 5 participants. Participants appreciated that the suggestions are made directly into their pull request and they can control over the decision of accepting or rejecting the suggestions. Furthermore, two participants that did not identify the wrong fix in the *Legacy* mode actually discovered it in the GitHub Suggestions mode. The participants explained that *GitHub Suggestions* allow them to focus more on reviewing the soundness of the fix, rather on understanding why the fix is there. Developers also appreciated the transparency, i.e., that the warning message was displayed in the suggestion box.

B. Interviews

Four participants described the experience with SAW-BOT as positive. One participant disagreed due to presence of wrong fixes: however, these fixes have been intentionally included in the study. Participants suggest to further improve the bot by displaying all SonarQube warnings as comments, even if SAW-BOT cannot propose fixes for those, as well as extending the list of warnings that can be fixed automatically.

While the clarity of *Pull Request* mode has been appreciated, the interviewees recognized such shortcomings as the large number of commits that might be created, or the lack of transparency regarding what each fix represents. Participants suggested to improve the bot by merging commits that refer to the same warning and occurring next to each other: e.g. the commented code violation is triggered for every three lines of commented code, leading for many warnings being reported for the same commented out fragment.

The *GitHub Suggestions* mode has been preferred by most participants due to the ease of workflow integration, detecting potentially wrong fixes and preventing new problems from entering the code. One participant mentioned that the GitHub Suggestions mode makes it easier for developers to understand their mistakes and learn from them. Another participant, however, disliked this mode, claiming that the suggestions placed next to the manually-written code might confuse developers. One improvement has been suggested, i.e., adding the GitHub Suggestions to solve legacy SonarQube issues (i.e. combining the Legacy mode with GitHub Suggestions).

C. mTAM

mTAM evaluates perceived usefulness and perceived ease of use [17]. Since one participant did not fill the questionnaire and one participant did not score all the items, we score the missing values as 4, following Lah et al. [17]. To ensure the reliability of the measurement, we compute the Cronbach's α [21], which is a metric used for evaluating the internal consistency of a set of items. The values of Chronbach's α range from 0 to 1. Values below 0.5 are interpreted as unacceptable, while a minimum between 0.65 and 0.8 is recommended [21]. The computed values are: for perceived usefulness α is 0.85, while for perceived ease of use it is 0.89. In other words, the responses are highly consistent.

SAW-BOT scored 79 on perceived usefulness and 71 on perceived ease of use, leading to the mTAM score 75. While Lah et al. do not present a way to interpret the mTAM scores, mTAM is meant to be similar to the System Usability Scale [22], and we base our interpretation on the corresponding guidelines [23]. This means that SAW-BOT ranks *acceptable* on the acceptability scale, C on the grade scale and *excellent* on the adjective rating scale [23].

V. Threats to validity

As any empirical study our work is subject to threats to validity. *Construct validity* refers to relation between the construct we would like to measure (developers' perception of SAW-BOT) and the measurements performed. To reduce the mono-method bias we combined three complementary approaches: the Wizard of Oz experiment, interviews and mTAM questionnaire.

To reduce the threats to *internal validity* we adopted established measurement instruments (mTAM), and followed best analysis practices (e.g., coding has been done by two persons). During the Wizard of Oz experiment, the first author did not mute their microphone when performing the Wizard's tasks. This has been done intentionally as muted microphone might have been perceived by the participants as a signal that the first author was doing something special, breaking the illusion of the existence of the bot. However, the very same illusion might have been threatened by the typing noise. Moreover, during the experiment the first author involuntarily disclosed that SAW-BOT is their bot. This could have made the participants adopt a more subjective stance, thus leading to a biased behavior.

Controlled environment of the Wizard of Oz experiment induces several threats to *external validity*. Participants have been working with a fairly simple Java project and the coding task had very explicit instructions, both reducing realism of the environment and potentially affecting generalizability of our findings to the actual working environment of Philips Research. Other factors that could potentially affect the generalizability of our findings include the small number of participants and the use of convenience sampling. We believe that evaluation of SAW-BOT in a more realistic environment should be subject of a follow-up study.

VI. Conclusions

In this work we have advocated the use of GitHub suggestions as a mechanism for bots proposing fixes for static analysis warnings. We have implemented SAW-BOT and shown that developers prefer GitHub suggestions to baseline operation modes, Legacy and Pull Requests. This being said, GitHub suggestions might not be uniformly beneficial for all kinds of software engineering bots. Understanding when GitHub suggestions are more and less beneficial, and what are the limitations of this approach should be subject of future work. Another direction for future work is the extension of SAW-BOT based on the suggestions made by the study participants. Moreover, since SAW-BOT could potentially benefit developers' productivity, subsequent studies could investigate if using SAW-BOT significantly impacts the development time. Finally, future studies could study the correlation of the preferred operation modes with the volume and criticality of the fixes.

VII. Data availability

The anonymized card sorting results and mTAM questionnaire, as well as the detailed Wizard of Oz experiment script and the post-experiment interview guide are publicly available [24].

At the moment of writing, the source of SAW-BOT is not public, but discussion about open-sourcing the project is ongoing. If a favorable decision will be made, SAW-BOT will be available on the Philips Software GitHub page.[7]

References

[1] C. Vassallo, S. Panichella, F. Palomba, S. Proksch, H. C. Gall, and A. Zaidman, "How developers engage with static analysis tools in different contexts," *Empir. Softw. Eng.*, vol. 25, no. 2, pp. 1419–1457, 2020.

[7] https://github.com/philips-software

[2] B. Johnson, Y. Song, E. R. Murphy-Hill, and R. W. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, D. Notkin, B. H. C. Cheng, and K. Pohl, Eds. IEEE Computer Society, 2013, pp. 672–681. [Online]. Available: https://doi.org/10.1109/ICSE.2013.6606613

[3] D. Marcilio, C. A. Furia, R. Bonifácio, and G. Pinto, "Spongebugs: Automatically generating fix suggestions in response to static code analysis warnings," *JSS*, vol. 168, p. 110671, 2020.

[4] C. Sadowski, J. van Gogh, C. Jaspan, E. Söderberg, and C. Winter, "Tricorder: Building a program analysis ecosystem," in *ICSE*. IEEE, 2015, pp. 598–608.

[5] H. Adzemovic, "A template-based approach to automatic program repair of sonarqube static warnings," Master's thesis, KTH, EECS, 2020.

[6] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "AVATAR: fixing semantic bugs with fix patterns of static analysis violations," in *SANER*. IEEE, 2019, pp. 456–467.

[7] R. Bavishi, H. Yoshida, and M. R. Prasad, "Phoenix: Automated data-driven synthesis of repairs for static analysis violations," in *ESEC/FSE*. ACM, 2019, p. 613–624.

[8] M. Wyrich and J. Bogner, "Towards an autonomous bot for automatic source code refactoring," in *BotSE*. IEEE, 2019, p. 24–28.

[9] S. Urli, Z. Yu, L. Seinturier, and M. Monperrus, "How to design a program repair bot? insights from the repairnator project," in *ICSE-SEIP*. ACM, 2018, p. 95–104.

[10] T. Muske and A. Serebrenik, "Survey of approaches for handling static analysis alarms," in *SCAM*. IEEE, 2016, pp. 157–166.

[11] "Java static code analysis rules," https://rules.sonarsource.com/java.

[12] V. Lenarduzzi, F. Lomio, H. Huttunen, and D. Taibi, "Are SonarQube rules inducing bugs?" *SANER*, pp. 501–511, 2020.

[13] J. D. Gould, J. Conti, and T. Hovanyecz, "Composing letters with a simulated listening typewriter," *Commun. ACM*, vol. 26, no. 4, p. 295–308, Apr. 1983.

[14] N. Dahlbäck, A. Jönsson, and L. Ahrenberg, "Wizard of Oz studies — why and how," *Knowledge-Based Systems*, vol. 6, no. 4, pp. 258–266, 1993.

[15] N. M. Fraser and G. Gilbert, "Simulating speech systems," *Computer Speech & Language*, vol. 5, no. 1, pp. 81–99, 1991.

[16] T. Zimmermann, "Card-sorting," in *Perspectives on Data Science for Software Engineering*. Academic Press, 2016, pp. 137–141.

[17] U. Lah, J. R. Lewis, and B. Šumak, "Perceived usability and the modified technology acceptance model," *International Journal of Human–Computer Interaction*, vol. 36, no. 13, pp. 1216–1230, 2020.

[18] F. D. Davis, "Perceived usefulness, perceived ease of use, and user acceptance of information technology," *MIS Quarterly*, vol. 13, no. 3, pp. 319–340, 1989. [Online]. Available: http://www.jstor.org/stable/249008

[19] S. Hudson, J. Fogarty, C. Atkeson, D. Avrahami, J. Forlizzi, S. Kiesler, J. Lee, and J. Yang, "Predicting human interruptibility with sensors: A wizard of Oz feasibility study," in *CHI*. ACM, 2003, p. 257–264.

[20] A. Wood, P. Rodeghero, A. Armaly, and C. McMillan, "Detecting speech act types in developer question/answer conversations during bug repair," in *ESEC/FSE*. ACM, 2018, p. 491–502.

[21] C. Goforth, "Using and interpreting cronbach's alpha," https://data.library.virginia.edu/using-and-interpreting-cronbachs-alpha/.

[22] J. Brooke, "SUS: a "quick and dirty" usability," *Usability evaluation in industry*, p. 189, 1996.

[23] A. Bangor, P. Kortum, and J. Miller, "Determining what individual SUS scores mean: Adding an adjective rating scale," *J. Usability Stud.*, vol. 4, pp. 114–123, 2009.

[24] D. Serban, B. Golsteijn, R. Holdorp, and A. Serebrenik, "SAW-BOT: Proposing Fixes for Static Analysis Warnings with GitHub Suggestions," Mar. 2021. [Online]. Available: https://doi.org/10.5281/zenodo.4599158