An Automated Approach to Detect Violations with High Confidence in Incremental Code using a Learning System

Radhika D. Venkatasubramanyam
Siemens Technology & Services Pvt. Ltd.,
Research and Technology Center India
Bangalore, India
91 80 33132069
radhika.dv@siemens.com

Shrinath Gupta
Siemens Technology & Services Pvt. Ltd.,
Research and Technology Center India
Bangalore, India
91 80 33135607
shrinath.gupta@siemens.com

ABSTRACT
Static analysis (SA) tools are often used to analyze a software system to identify violation of good programming practices (such as not validating arguments to public methods, use of magic numbers etc.) and potential defects (such as misused APIs, race conditions, deadlocks etc.). Most widely used SA tools perform shallow data flow analysis with the results containing considerable number of False Positives (FPs) and False Negatives (FNs). Moreover it is difficult to run these tools only on newly added or modified piece of code. In order to determine which violations are new we need to perform tedious process of post processing the SA tool results. The proposed system takes into consideration the above mentioned issues of SA and provides a lightweight approach to detection of coding violations statically and proactively, with high degree of confidence using a learning system. It also identifies the violations with a quality perspective using the predefined mapping of violations to quality attributes. We successfully implemented a prototype of the system and studied its use across some of the projects in Siemens, Corporate Technology, Development Center, Asia Australia (CT DC AA). Experimental results showed significant reduction in time required in result analysis and also in FPs and FNs reported.

Categories and Subject Descriptors
D.2.9 [**Management**]: Software Quality Assurance

General Terms
Measurement, Performance, Reliability, Experimentation, Security, Human Factors, Standardization, Languages, Verification.

ICSE Companion'14, May 31 – June 7, 2014, Hyderabad, India
ACM 978-1-4503-2768-8/14/05
http://dx.doi.org/10.1145/2591062.2591123

Keywords
Code quality, code assessment, static analysis, software quality, false positives, learning system

1. INTRODUCTION
SA tools report huge number of violations for large software systems and most of the time programmers are only interested in the issues which are new compared to the previous version, particularly in applications having legacy code components [12]. Hence constructing warning deltas between versions, showing which issues are new, which have persisted, and which have disappeared is important. This allows reviewers to focus their efforts on inspecting new issues. Running SA tools on each new version and then post-processing to determine newly introduced issues can be extremely time consuming and may not be accurate. Reports generated are lengthy and considerable amount of time needs to be spent analyzing the reports, mainly due to the large number of FPs.

It would be faster and more helpful to the programmers if the issues could be proactively identified in the IDE itself. In addition to identifying issues, classify them based on various factors like its severity, its impact to the various quality attributes, FP rate etc. Since the same set of bugs could reappear in the new modules of the software system, there is a need to learn from these errors and identify potential defect areas in the new modules without the overhead of running SA tools.

Tracking warnings through a series of software versions reveals where potential defects are commonly introduced and addressed, and how long they persist; thus exposing interesting trends and patterns. This helps in determining which SA rules are important for a software system and helps select a minimum set of rules that must be enabled. In some cases, it becomes necessary to remember decisions about code that has been reviewed and found to be safe despite violations being reported by SA tools. This takes into consideration that some of the reported violations were FPs or not good to fix from the perspective of the software system. In other words, there is a need to determine whether warnings reported in multiple versions correspond to the same SA rule and to identify the list of SA rules that are violated more frequently and/or more recently.

What is essential is a system that can statically detect bugs in code with the scope being only the new set of code and not have the overhead of running multiple tools on the entire code base. The report generated should have violations or bug indications, provided as a prioritized listing, related to the newly added part of

code. Reporting of violations that are FPs had to be reduced. FNs should also be handled efficiently. We believe that if the tool can learn about FP and the FN patterns from the expert, the same patterns can be avoided in future.

The proposed system attempts to provide a light weight single step solution to error identification with higher accuracy by reducing both FPs and FNs. This is achieved through a learning system which runs on only the differential or the newly added/modified part of code.

1.1 Why Do We Need Proposed Method When We Have Static Analysis

SA is one of the most important techniques to identify coding violations. The system proposed in this paper is an attempt to supplement this process. The need is justified by several reasons, some of which are discussed below.

Multiple tools. To perform comprehensive error detection, more than one tool is required. Deep static analysis tools [11] are very expensive (e.g. Coverity [2]) to be procured. The learning curve to gain knowledge of these tools is high and using multiple tools on a daily basis demands considerable amount of time and resource. Added to this is the overhead involved in analyzing the large list of violations reported in the tool outputs, and merging them into a single set of findings.

Differential code. SA tools typically need to be run on the entire source code or intermediate representation of the code and usually cannot be run on only the selected or newly added parts of code. It is also difficult to filter and identify the violations that occur only in the newly added/modified part of the code.

Lack of tools that learn from experts. To the best of our knowledge there are no SA tools which can learn from the experts. Considering this, we propose using a learning system that learns based on several criteria described in the following section.

False Positives. Identification of FPs in the tool reports involves inspecting the list of violations and verifying in the code if it is really an issue. Given the tight deadlines in project development, such detailed verification of each of the issues poses overheads with respect to time and resource. In some cases, FPs are marked or annotated in the tool outputs and these annotations are noted for future reference.

False Negatives. Tools, in order to increase the precision level (by reducing FPs) tend to miss true defects.

One solution to avoid FPs and FNs (as proposed in this paper) would be to use expert knowledge. An expert, while validating the violations from SA tools, can identify the patterns and contexts in which these patterns occur. This information can be input into a learning system, using which subsequent analyses could be performed.

2. OVERVIEW OF THE METHODOLOGY

The proposed system finds accurate and actionable issues by performing analysis of code across versions at the time of commit of each change in the new version, by using a learning system. Hence this is a proactive method to identify issues which could be detected at a later stage after running static analysis and metric tools. In addition, the system could predict issues and classify them based on several factors such as confidence level, criticality

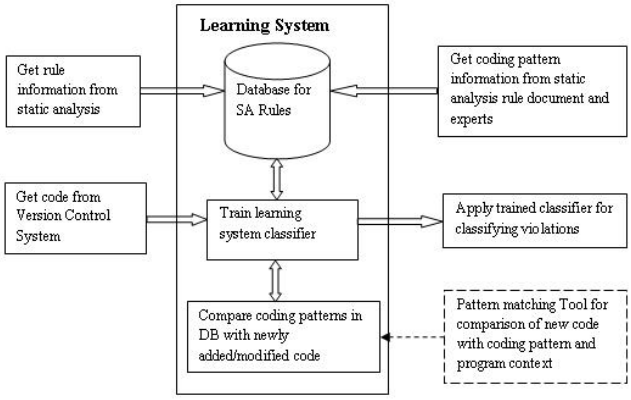

Figure 1. Overview of the proposed system

and impact of the rule on code with relevance to quality attributes, most frequently violated, and most recently violated rules.

The overview of the methodology is as shown in Figure 1. As indicated in the figure, the learning system takes information about coding patterns from static analysis rule documentation and static analysis report of the source code of the system under study.

The program context under which a rule can lead to FPs or FNs is also added. The training set for the learning system is built using these information. In parallel, the database that stores information about static analysis rules is continuously updated. The information stored in the database also includes user input about coding pattern that can lead to FNs and coding patterns corresponding to true positives.

We use the following criteria for decisions on selection and filtering SA rules that detect coding violations:

- Patterns of code where SA violations are reported
- Impact of violations on the quality of the code – This takes into account the side-effect of the violation by observing the quality attribute it affects
- Confidence level of the rules – This filters the rules to be provided to the learning system based on probability of rule to report FPs.
- Most commonly committed errors reflecting the programmer's pattern of coding
- Most recently committed errors

A pattern matching tool is used for comparing modified or newly added piece of code with the coding patterns. Information on the timestamp of the code (most recently modified or newly added) is obtained from Version Control System (VCS).

Once the learning system is trained it can predict issues in the newly added or modified code and present the output to the user in the prioritized format. Prioritization can be performed using factors such as important quality attributes for the project, impact of violations, recurring or a new violation etc.

2.1 Formulating the Learning System

This section outlines the steps involved in realizing the proposed technique. An initial requirement is that of a database which will contain the mapping of SA rules to coding patterns selected based on some filtering criteria. The database can be modified based on user experience as and when he/she finds a new coding pattern and other filtering criteria.

Step 1 – Obtaining initial training set
The first step consists of obtaining the initial training set for the learning system. It involves running SA tools on 'k' most recent versions of the system. 'k' the number of versions is defined as:

k = 3; *If Number of Versions < 10*

k = (num. of versions / 3); *If Number of Versions > 10*

Source code of the most recent 'k' versions is obtained from the source code repository and SA tools are run on each version. For versions which have SA tool reports available, there is no need to run SA tools again and reports can be directly used.

The heuristic behind choosing most recent 'k' versions is that recently violated rules will be violated again. At the same time the sample size should be large enough so that rules can be effectively chosen, hence we are using the above mentioned formula.

Step 2 - Listing of violated SA rules
The SA tool reports obtained are post-processed to determine list of SA rules which are violated. The list serves as a catalog from which rules can be selected and added to the learning system.

Step 3 - Rule selection
Rules are filtered based on some filtering criteria and added to the learning system database. The selected list consists of rules based on its relevance and occurrence in the software module being reviewed.

Step 4 - Updating rules in database
The learning system will keep learning and keep updating the database which contains rule id and coding patterns violation based on the selected set of rules. It will also contain the coding pattern where violation of a rule is a FP or FN. These coding patterns can be entered by experts and they can modify it on continuous basis, if required.

Step 5 - Check if Release is an Intermediate Release
This step checks for the type of release. Depending on the type the respective next steps are taken.

- If the release is an internal/intermediate release, then the VCS is queried to obtain the part of code which is new or modified.
- If the release is not an internal one, then run SA tools on the entire code base and go to step 2.

Step 6 - Comparison of new or modified code against code patterns in learning system
Every change made and saved by the programmer is checked against the learning system corpus and possible bugs can be identified immediately. A pattern matching tool can be used to compare the new code with code patterns present in the learning system database. If any matches are found, then there is a violation of the rule whose code pattern matched with the newly introduced or modified code snippet. To improve accuracy of results the learning system will also learn about code patterns where a rule can give FPs.

Step 7 - Get comparison results
The comparison results provide data to check against factors of MFV (Most Frequently Violated) and MRV (Most Recently Violated). The learning system database is modified (rules are added or removed) based on these factors.

3. INITIAL EXPERIMENTAL RESULTS

We studied three projects from the Siemens CT DC AA Healthcare sector. The programming language used is C++ and the size of the code base ranged from 200 KLOC to 500 KLOC. We used few sophisticated SA tools on these projects and found that for critical rules the FP rate was about 35%. We also observed several occurrences of FNs which were detected only by code inspections through manual review.

The code in the selected projects was analyzed using this prototype. We observed a reduction of about 15% in FPs reported. We also trained the system on detecting FNs and observed that it detected considerable number of FNs learnt from expert knowledge. The results obtained using our prototype showed a 24% reduction in time spent in analyzing the results.

In the course of prototype implementation, we also analyzed the reasons for occurrence of FPs and FNs and observed that one of the reasons is involved difficulty for a SA rule implementer to anticipate all the patterns leading to their occurrences. Anderson [9] has observed that conservative analysis is a factor leading to FPs. Some of the other observed reasons leading to FPs are mentioned below:

- Bugs in SA rule implementation
- Lack of documentation of APIs which may lead to improper interpretation of the APIs of SA tools while writing custom rules

To provide more insight into how patterns about FPs and FNs are captured, consider the following two examples leading to FPs and FNs in rules.

- Consider the rule: "*Assignment "=" is used within a "if" statement or Assignment "=" is used within "while" statement* " as R1. Then the following first-order predicate logic formula demonstrates program context under which it can lead to **FPs**.

Here, *S* represents the set of all statements of the program. This occurs when an assignment and equality check operators are both encountered in the if/while condition statement.

$$\forall \boldsymbol{S} \begin{bmatrix} ((\boldsymbol{S} \in \boldsymbol{IfStmtSet})\ \boldsymbol{OR}\ (\boldsymbol{S} \in \boldsymbol{WhileStmtSet}))\ \boldsymbol{AND} \\ (\boldsymbol{S\ has\ AssignmntOptr}) \\ \boldsymbol{AND}\ (\boldsymbol{S\ has\ EqualityOptr}) \\ \rightarrow\ \boldsymbol{R1\ leads\ to\ FPs} \end{bmatrix}$$

- Consider the rule "Avoid empty constructs" as *R2*. Then the following first-order predicate logic formula demonstrates program context under which it can lead to **FNs**.

Here, *s* represents the set of all statements belonging to method m. The case where the return type of the method is void and it contains only empty conditional statements, then the method is considered a "*do nothing*" method. But the check might not detect the empty method due to the presence of the conditional statements.

$$\forall \boldsymbol{m}, \boldsymbol{s} \begin{bmatrix} (\boldsymbol{s} \in \boldsymbol{Method(m)}\ \boldsymbol{AND} \\ (\boldsymbol{s\ is\ EmptyConditionalOrLoopingStmt}) \\ \boldsymbol{AND}\ (\boldsymbol{ReturnType(m) is\ Void}) \\ \rightarrow \boldsymbol{R2\ leads\ to\ FNs} \end{bmatrix}$$

Similarly, context information for other rules that were violated in the projects under study was obtained and fed into the prototype implementation of the proposed learning system as a part of the training set. The severity and impacted quality attribute

information were also extracted for each rule that was to be added to the learning system.

4. RELATED WORK

Sunghum et al. [1] have proposed change classification for predicting latent software bugs. Their approach, using a machine learning classifier, helps identify bug prone statements at the time changes were made; however, the training set is created by extracting features from changes obtained from SCM systems.

Spacco et al. [3] propose the two techniques of *pairing warn/ngs* and *warning signatures* for tracking defect warnings across versions. These approaches may not give correct results when two findings belong to same method. In our approach we not only identify the new warnings but also make the system learn about FPs and FNs.

Guangtai et al. [4] show the importance of an effective training set for warning prioritization and propose an automatic approach to construct an effective training set by mining generic-bug-fix history. This approach however does not consider the importance of a static analysis rule and its impact on quality attributes for prioritizing the warnings.

Sliwerski et al. [2] have proposed that fix-inducing changes can be identified by linking a version archive to a bug database. A history based warning prioritization scheme is proposed by Kim et al. [5], which mines for removed warnings during bug fixes. The approach mentioned by Bell et al. [6] uses age of the file as an important factor to predict which files in a large industrial software system could contain faults in the next release. Developer feedback is important for identifying FPs, as done by Heckman et al. [7] (using adaptive models). They similarly consider developer's feedback for determining FPs, but they do not consider other important criteria for prioritizing warnings.

Apiwattanapong et al. [8] performs program differencing by performing graph isomorphism, which has NP-Complete complexity. The scope of comparison is the structure of the code in the two versions. Multiple versions and impact on quality is not considered in this method.

Several techniques to analyze and post process the static analysis results have been proposed. But none of the existing techniques address the creation of the training set based on static analysis results. All the current methods are static in the sense that they use only fixed algorithms, with no learning from the previous versions or expert knowledge/experience. The additional information of the prospective impact of the bug on the quality attributes, as proposed in this paper, will help the programmer realize its criticality and take relevant necessary steps to correct the statements and also facilitate proactive monitoring of bugs.

5. SUMMARY

In this paper, our attempt is not to replace usage of static analysis completely. But to augment the existing review process, by reducing the overheads of running these SA tools on a daily basis. Also, to provide the programmer with code analysis and potential bug information, taking into account the quality attributes, confidence level, and other user related criteria that helps filter and select the most relevant rules and violations. If SA tools can be run less frequently (for example, every major release only, avoiding SA analysis for minor releases) and only on the newly added code, time spent in running the tools can be greatly reduced. Since the size of the code set being scrutinized is small, the reports will be more manageable and easier to fix.

With experience and code review expertise, the database of code patterns can be extended to include more patterns, thus increasing the usefulness of the proposed system. There are some deep static analysis problems that cannot be expressed efficiently as coding patterns because they are either extremely context-specific or may require inter-procedural static analysis. For such kind of problems use of first order predicate logic to express them in our learning system can improve accuracy further and analysis should be performed on the changed piece of code (and dependent code) so that analysis time is within reasonable limits. We believe that over time, learning based classifier may introduce subjectivity of experts. Due to misinterpretation, the experts may mark a serious/hard-to-understand issue as FP; control must be exercised to avoid this.

6. REFERENCES

[1] Sunghun Kim, Sunghun, E. James Whitehead, and Yi Zhang. "Classifying software changes: Clean or buggy?." Software Engineering, IEEE Transactions on 34.2 (2008): 181-196.

[2] Śliwerski, Jacek, Thomas Zimmermann, and Andreas Zeller. "When do changes induce fixes?." ACM sigsoft software engineering notes 30.4 (2005): 1-5.

[3] Spacco, Jaime, David Hovemeyer, and William Pugh. "Tracking defect warnings across versions." Proceedings of the 2006 international workshop on Mining software repositories. ACM, 2006.

[4] Liang, Guangtai, et al. "Automatic construction of an effective training set for prioritizing static analysis warnings." Proceedings of the IEEE/ACM international conference on Automated software engineering. ACM, 2010.

[5] Kim, Sunghun, and Michael D. Ernst. "Which warnings should I fix first?." Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering. ACM, 2007.

[6] Bell, Robert M., Thomas J. Ostrand, and Elaine J. Weyuker. "Looking for bugs in all the right places." *Proceedings of the 2006 international symposium on Software testing and analysis*. ACM, 2006.

[7] Heckman, Sarah Smith. "Adaptively ranking alerts generated from automated static analysis." Crossroads 14.1 (2007): 7.

[8] Apiwattanapong, Taweesup, Alessandro Orso, and Mary Jean Harrold. "A differencing algorithm for object-oriented programs." *Proceedings of the 19th IEEE international conference on Automated software engineering*. IEEE Computer Society, 2004.

[9] Anderson, Paul. "The use and limitations of static-analysis tools to improve software quality." CrossTalk 21 (2008): 18-21.

[10] Coverity. http://www.coverity.com/ [Online; accessed on 22-November-2013].

[11] Emanuelsson, Pär, and Ulf Nilsson. "A comparative study of industrial static analysis tools." *Electronic notes in theoretical computer science* 217 (2008): 5-21.

[12] Lahiri, Shuvendu K., Kapil Vaswani, and C. AR Hoare. "Differential static analysis: opportunities, applications, and challenges." *Proceedings of the FSE/SDP workshop on Future of software engineering research*. ACM, 2010.