Reducing Human Effort and Improving Quality in Peer Code Reviews using Automatic Static Analysis and Reviewer Recommendation

Vipin Balachandran
VMware
Bangalore, India
vbala@vmware.com

***Abstract*—Peer code review is a cost-effective software defect detection technique. Tool assisted code review is a form of peer code review, which can improve both quality and quantity of reviews. However, there is a significant amount of human effort involved even in tool based code reviews. Using static analysis tools, it is possible to reduce the human effort by automating the checks for coding standard violations and common defect patterns. Towards this goal, we propose a tool called Review Bot for the integration of automatic static analysis with the code review process. Review Bot uses output of multiple static analysis tools to publish reviews automatically. Through a user study, we show that integrating static analysis tools with code review process can improve the quality of code review. The developer feedback for a subset of comments from automatic reviews shows that the developers agree to fix 93% of all the automatically generated comments. There is only 14.71% of all the accepted comments which need improvements in terms of priority, comment message, etc. Another problem with tool assisted code review is the assignment of appropriate reviewers. Review Bot solves this problem by generating reviewer recommendations based on change history of source code lines. Our experimental results show that the recommendation accuracy is in the range of 60%-92%, which is significantly better than a comparable method based on file change history.**

I. Introduction

Static analysis techniques work on a source representation of the software with the goal of finding defects early in the development. The peer code review process, where a programmer presents his/her code to one or more colleagues, is one of the most commonly used static analysis technique [1] and has many advantages [2]. Code review is often considered as a cost-effective defect detection technique due to the early detection of bugs, when it is less expensive to fix [3].

There are different ways to perform a code review including the most formal *code inspection* to the least formal *over-the-shoulder* review [4]. Tool assisted code review uses software assistance in different phases of the review and reduces the large amount of paper work, which code inspections require. In contrast to code inspection, tool assisted code review supports distributed reviews and improves both quality and quantity of reviews [5].

The main challenge in code review is the significant amount of human effort involved; this is true even for tool assisted code reviews. For example, in Mozilla projects, every commit to the repository has to be reviewed by two independent reviewers [6]. In VMware, most of the projects require at least two reviewers for every commit. Another challenge in code review is the learning curve involved in understanding the defect patterns [7] and coding conventions to check for [2]. The experimental results in Section IV-A2 show that there are plenty of coding standard violations in approved code reviews. As mentioned in [2], this could be due to either excessive amount of information involved in enforcing the coding standard or the reviewers deprioritizing these checks in favor of logic verification. We believe that checking for coding standard violations during code review is as important as defect detection due to the potential benefits of a consistent coding style [2], [8], [9].

Another challenge in code review is to assign appropriate reviewers. The reviews would be time consuming or inaccurate if appropriate reviewers are not set. Mozilla projects require that at least one of the reviewers must be the module owner or the module owner's peer [6]. In VMware, there is no company-wide policy regarding the assignment of reviewers. However, many projects do follow conventions similar to Mozilla projects. The challenge here is to identify a module owner, especially in the case of large projects spanning multiple geographies, where multiple developers making changes in different parts of the same file is normal. Most of the time, novice developers have to reach out to experienced developers or search the file revision history to assign appropriate reviewers.

Automatic static analysis using static analysis tools is much faster and cheaper than code reviews and quite effective in defect detection [1], [10], [11]. Using static analysis tools, it is possible to automate the checks for coding standard violations and common defect patterns. In this paper, we introduce a tool called Review Bot aimed to address the problems discussed above with tool assisted code review. Review Bot is built as an extension to Review Board [12], an open-source code review tool, which has many features including file gathering, diff display, comments collections, threaded discussions, etc. Review Bot uses multiple static analysis tools to automate the checks for coding standard violations and common defect patterns, and publish code reviews using the output from these tools. This will reduce the reviewer effort, since there is no need to spend time on style related checks and common defect

978-1-4673-3076-3/13 © 2013 IEEE

ICSE 2013, San Francisco, CA, USA
Software Engineering in Practice

Accepted for publication by IEEE. © 2013 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

patterns. The reviewers can pay more attention to logic verification rather than on checks which can be automated. This would improve the reviewer productivity and result in better quality reviews. Even though, the current implementation of Review Bot supports only Java, the architecture is generic to support any programming language with static analyzers available. Section III-C2 discusses on extending Review Bot to add support for other programming languages.

To solve the reviewer assignment problem, we propose a reviewer recommendation algorithm as part of Review Bot. The recommendation algorithm is based on the fact that the most appropriate reviewers for a code review are those who previously modified or previously reviewed the sections of code which are included in the current review.

It may be argued that developers themselves can run the static analysis tools in their development environment before creating a review request or before committing the changelist into the repository (depot). This may not be effective, since there is no administrative control over the process. It is also difficult to apply common configuration for static analysis tools, when used in a decentralized manner. As mentioned in [10], the static analysis tools may receive limited use depending on how they are integrated into the software development process. We believe that the defects can be detected much early in the development process if automatic static analysis is tightly integrated with the code review process.

The main contributions of this paper are as follows:

- We propose a tool based on automatic static analysis to reduce the human effort in code reviews.
- Through a user study, we show that tightly integrating static analysis tools with the code review process could improve the quality of code review.
- We propose a recommendation algorithm for solving the reviewer assignment problem, which would ease the task of reviewer assignment in large projects.

The remainder of this paper is structured as follows. In Section II, we discuss briefly about Review Board, the code review process in VMware and the various static analysis tools used in Review Bot. The Review Bot architecture and algorithms for automatic review generation and reviewer recommendation are described in Section III. In Section IV, we present the results of a user study and the experimental results of the reviewer recommendation algorithm. The related work is discussed in Section V and we conclude in Section VI.

II. Background

A. Review Board

Review Board is an open-source web-based code review tool. Automation is supported using REST API. The requests are made to resources, which are locations containing data, encoded in JSON or XML.

Terminology: The relationship between different Review Board resources is shown in Fig. 1. A *filediff* resource provides information on a single, self-contained raw diff file that applies to exactly one file in a code repository. A list of filediff

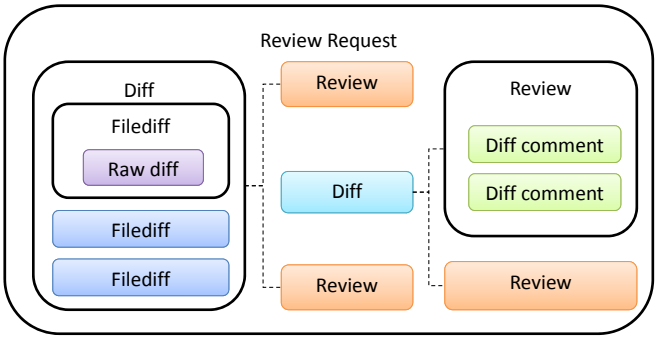

Fig. 1. Relationship between various Review Board resources

resources constitutes a *diff* resource and a list of diff resources constitute a *review request* resource. Each diff in a review request is revisioned, with the revision starting from 1. Each of the code reviews of a review request, published by a reviewer is represented by a *review* resource and is associated with a single diff resource. The review resource contains a list of *diff comment* resources. A diff comment resource provides information on a single comment made on a filediff.

For example, assume that the developer's changelist contains two modified files. A review request resource is created by uploading the raw diff files corresponding to all the files in the changelist. The two raw diff files result in two filediff resources, which constitute a diff resource with revision number 1. When a code reviewer enters some comments (represented by diff comment resources) against the changed lines after inspecting the changes in the diff view, a review resource is created. The developer uploads the latest raw diff files after addressing the comments, which creates a new set of filediff resources and hence, a new diff resource (with revision number 2).

B. Code Review Process

In this section, we outline the code review process (Fig. 2) followed by most of the project teams in VMware. The various steps in the process are as follows:

1) User (submitter) generates raw diff data of all the files in the changelist.
2) Submitter creates a new review request in Review Board.
3) Submitter enters review request description, uploads the raw diff files, assigns reviewers and posts the review request.
4) All the reviewers will receive an email from Review Board notifying the review request.
5) The reviewer logs into Review Board and selects the diff view of the filediffs in the latest diff revision.
6) The reviewer inspects the code and enters comments against the lines displayed in the diff view using the add comment option. Even though, Review Board allows concurrent reviews of a single diff revision, the submitter optionally sends emails to designate reviewers as primary and secondary to prevent redundant reviews. The primary reviewer begins the review first while the secondary

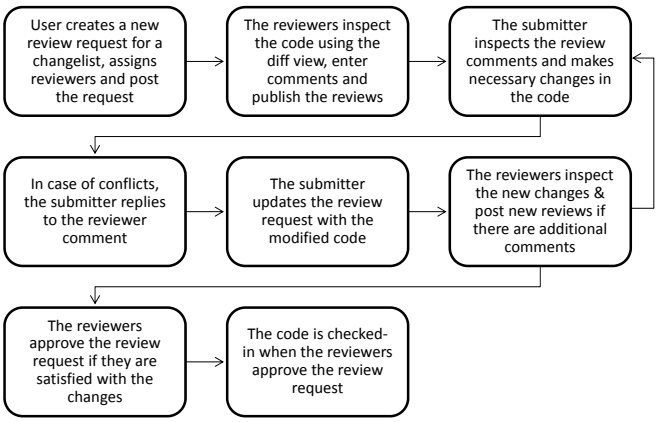

Fig. 2. Code review process

reviewer starts the review only after the primary reviewer approves the review request.

7) When the inspection is over, the reviewer publishes the review.
8) Review Board sends an email to the submitter and all the reviewers notifying the new review.
9) The submitter logs into Review Board and inspects the review comments.
10) If the submitter accepts the comment, necessary changes are made in the workspace version of the source file. In case of conflicts, the submitter replies to reviewer's comment.
11) If changes are made in one or more files in the changelist, the submitter generates new raw diff data and uploads it in Review Board, which creates the next diff revision.
12) All the reviewers will receive an email notifying the review request update.
13) The reviewer can view the changes made between the current diff revision and the previous revision using the diff viewer's inter-diff view.
14) If there are any unaccepted comments, the reviewer can provide more details by replying to the submitter's comment.
15) If the reviewer is satisfied with the changes, he/she can do another pass or approve the review request by clicking the *approve* button.
16) When the review request is approved by at least two reviewers, the submitter can check-in the changelist into the repository.

C. Static Analysis Tools

This section discusses the static analysis tools used in the current implementation of Review Bot.

Checkstyle: Checkstyle [13] is a static analysis tool for Java, which checks for coding standard violations in source code. The various checks are represented as checkstyle modules, which can be configured to suit a particular coding style. Checkstyle can be extended by writing user-defined checks in Java.

PMD: PMD [14] scans Java source code and looks for potential problems like possible bugs, dead code, suboptimal code, overcomplicated expressions and duplicate code. The checks are represented by rules, organized under different rule sets; for e.g., the *Code Size* rule set contains rules that find problems related to code size or complexity. PMD can be extended by writing new rules in Java or XPath.

FindBugs: FindBugs [15], [16] works by analyzing Java byte code, searching for predefined bug patterns. A bug pattern is a code idiom that is often an error. There are 380 bug patterns grouped in different categories such as Correctness, Bad Practice, and Security [10]. FindBugs can be extended by adding new bug detectors written in Java.

A detailed comparison of various static analysis tools for Java can be found in [17]. The selection of these tools was motivated by their popularity, ease of extensibility and the problem areas covered. While Checkstyle covers coding style related issues, PMD checks class design issues and questionable coding practices. FindBugs, on the other hand, detects potential bugs in the code.

III. Review Bot

Review Bot is a stand-alone Java application, which can generate automatic reviews (in Review Board) and reviewer recommendations. When used, it will be treated as a normal code reviewer and often reviews the code first. As part of the review, it also recommends human reviewers appropriate for reviewing the current request.

A. Modified Code Review Process Using Review Bot

In this section, we outline the changes required in the existing code review process (Section II-B) in order to introduce Review Bot. The changes are as follows:

- In step 3, the submitter enters Review Bot as the reviewer and posts the review request.
- Review Bot creates draft auto-review (Section III-C) for the latest diff revision.
- Review Bot generates reviewer recommendations (Section III-D) for the latest diff revision.
- Review Bot posts auto-review with the review summary containing top-3 recommended reviewers.
- Submitter addresses the comments which he/she agrees, drops the remaining comments and generates the next revision of the diff.
- Review Bot posts an automatic review for the latest diff revision to verify whether the submitter has fixed all the accepted comments and no new issues are introduced.
- Once all the comments in automatic reviews are either fixed or dropped, the submitter removes Review Bot and enters the recommended reviewers in the reviewer list.
- Continue with step 4 in the current code review process.

It is also possible to perform auto-review at any time by adding back Review Bot as the reviewer. This is recommended especially when the submitter introduces new files or makes a lot of changes in existing files as part of addressing manual review comments.

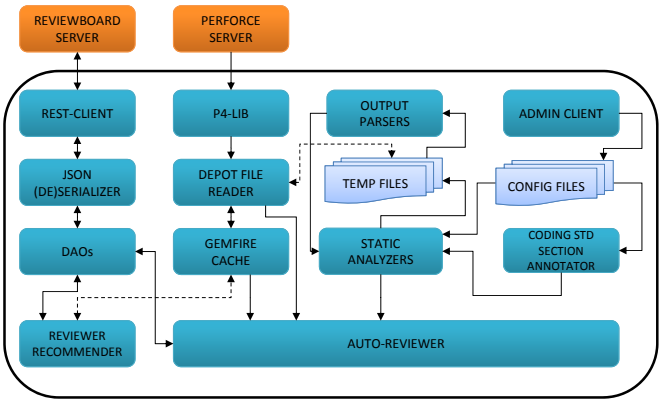

Fig. 3. Review Bot architecture

B. Architecture

Fig. 3 shows the Review Bot architecture. Review Bot connects to Review Board through the Rest Client built using Jersey library [18]. The JSON serializer/deserializer based on Jackson [19] is responsible for the conversion between Java objects representing Review Board resources such as Review Request and the corresponding JSON (one of the wire formats of Review Board) representation. The Data Access Objects (DAOs) are used to read/write Review Board resources. GemFire [20] cache is an in-memory disk persisted cache and is used to cache and persist data such as Review Board resources, meta-data about raw diff files and depot files cached in the local file system. The admin client provides a web interface to support project specific configuration of static analysis tools, which includes enabling or disabling checks/rules, modifying rule settings and messages, and mapping rules to relevant sections in coding standard.

C. Automatic Review Generation

Algorithm 1 *AutoReview*(id, rev, $proj$)

```
1:  // id: Review Request ID, proj: project name, rev: diff revision
2:  ReviewRequest req ← getReviewRequest(id)
3:  Diff diff ← req.getDiff(rev)
4:  StaticAnalyzer[ ] sas ← getStaticAnalyzers(proj)
5:  List⟨SourceCodeIssue⟩ issues ← { }
6:
7:  for (FileDiff fileDiff : diff.getFileDiffs()) do
8:    if (isFileTypeSupported(fileDiff, proj)) then
9:      File f ← getMergedFile(fileDiff)
10:     for (StaticAnalyzer s : sas) do
11:       issues.addAll(s.check(f, proj))
12:     end for
13:   end if
14: end for
15:
16: Review review ← createReview(req, issues)
17: review.publish()
```

The auto-review procedure is outlined in Algorithm 1. In lines 2-3, the review request and the diff to be reviewed are read and assigned to req and $diff$ respectively. The array sas is initialized to static code analyzers configured for the given project in line 4. The **for** loop in lines 7-14 iterates over all the file-diffs in $diff$ and invokes the static code analyzers to find source code issues in each of them. The **if** statement in line 8 ensures that non-supported file types for the project are skipped. The $getMergedFile$ function call in line 9 checks whether the depot file and the patch file (raw diff data) corresponding to the given filediff are already present in the local file system based on the metadata stored in gemfire and if not, downloads these files from respective servers (depot file from revision control system and patch file from Review Board server) and updates the metadata. Once the depot file and patch file are downloaded, the *patch* [21] program is used to create the merged/patched file.

Each of the static code analyzers configured for the project is invoked with the corresponding project specific configuration. The output from these static analyzers is parsed into a common format representing the source code issues, and finally, these issues are annotated with relevant coding standard section numbers. These steps correspond to the $check$ function call in line 11. There is one output parser required per static analyzer, which understands the format of corresponding static analyzer output. All of the static analyzers used in the current implementation supports XML as one of the output formats and output parsers can be easily written. The common format representing a source code issue has two mandatory fields: *i*) the starting line number in the source code where the issue is found and *ii*) the warning message. The output parsers can also populate optional fields such as the ending line number, the starting and ending column numbers, numerical priority in the range 1...5 and rule code, if the data is available in the static analyzer output.

The $createReview$ function call in line 16 creates a draft review with comments corresponding to the source code issues. Finally, in line 17, the draft review is published.

1) Support for Static Analyzers Operating on Object Code or Byte Code: FindBugs, unlike Checkstyle or PMD requires Java byte code rather than source code as input. Since the source files in the diff can have dependencies which may not be part of the diff, it is not always possible to compile the merged source files to generate the byte code or object code. The auto-reviewer component can solve this by maintaining a local copy of the project source code. When the byte code or object code is required, the local copy is first synchronized to the changelist in the repository to which the submitter's workspace is synced, then the merged files are copied and finally, the project is built. To the best of our knowledge, there is no direct way to obtain the changelist to which the submitter's workspace is synced, other than prompting the submitter for it and including it as part of the review request. However, for our experiments, we determined this changelist by trial-and-error, as discussed below.

The filediff contains the revision number of the repository file to which the raw diff data applies, known as the base revision. The revision control system is queried to retrieve the list of changelists which created the *base revision* and *base revision+1* of files corresponding to various filediffs. Let these lists be CL_{prev} & CL_{next} respectively and defined as:

$$CL_{prev} = \langle cl_i, cl_{i+1}, ..., cl_{i+m} \rangle$$

$$CL_{next} = \langle cl_j, cl_{j+1}, ..., cl_{j+n} \rangle$$

where the changelists in CL_{prev} and CL_{next} are arranged in the chronological order of check-in time. It is clear that the changelist to which the submitter's workspace is synced is always between the cl_{i+m} (inclusive) and clj (exclusive). Once this possible list of changelist numbers is determined, the local workspace is synced to each of these changelists one-by-one, until the build succeeds. However, this method will not work if all the filediffs correspond to new files. In such cases, the possible list of changelist numbers is determined based on the last update time of the review request. The submitter's workspace couldn't have been synced to any changelist submitted after the last update time of the review request.

2) Extending Automatic Review: This section discusses how to extend automatic review generation for programming languages other than Java. Adding support for a new language includes three steps: *i*) configuring one or more static analyzers for the language; *ii*) writing an output parser for each of the static analyzers; and *iii*) mapping each of the checks/rules enabled in the static analyzer to the corresponding section (if any) in the coding standard. For example, Python support can be added by configuring auto-reviewer to use Pylint [22] if the source language is Python. The output from Pylint should be converted to the common format before generating the auto-review, which requires a new output parser. The default rule settings of Pylint may need tweaking to enforce the desired coding standard. The mapping of the Pylint rule codes to the corresponding coding standard sections enable the coding standard section annotator to annotate the auto-review comments with section numbers.

Review Bot requires the static analyzer to satisfy four requirements: *i*) it should be possible to execute the static analyzer in a sub-process; *ii*) the output from the static analyzer should be able to redirect to a file, which can be read by the output parser; *iii*) the output should have a well defined structure so that a parser can be written; and *iv*) each of the source code issues reported should contain a line number indicating its position in the source code and a warning message. Most of the popular static analysis tools in the open domain can be called from an external program, and their output can be redirected to a file. They also support output reporting in multiple formats including XML.

3) Discussion on Usability: The integration of static analysis warnings in code review, as discussed above, has some usability issues, which need to be addressed. The first issue is related to the presentation of warnings in a user friendly manner. Review Board has built-in support for presenting the review comments in the side-by-side diff view. Since an automatic review is no different from a manual review, its presentation is automatically taken care by Review Board. Refer Section IV-A3 for screenshots of some of the diff comments in automatic reviews, as presented by Review Board.

The second usability issue is related to the comments which the submitter has dropped using the *drop comment* option in the diff-viewer. If care is not taken, the same comment will be reported again when the user requests auto-review for the next diff revision. The auto-reviewer solves this by cross checking any comment generated against similar comments (with matching line number) in previous auto-reviews and refrain from reporting, if it was dropped previously.

The static analyzers generate a large number of false positives [17], which poses another usability issue. Since the auto-reviewer component of Review Bot depends on static analyzers, the large number of false warnings may hinder its adoption in the code review process. We try to reduce the false positives by enabling only a subset of checks and configuring the rule settings.

D. Reviewer Recommendation

The reviewer recommendation helps review request submitter to assign most appropriate reviewers and reduce the time taken for review acceptance. In large software projects, the review request submitter often assigns reviewers based on the revision history of files in the diff. Since a file could be edited by multiple developers, it is a difficult task to assign appropriate reviewers if the number of files involved and the changes within them are large. To ease the selection of reviewers, we propose an algorithm based on *line change history* of source code for generating reviewer recommendations. The line change history of a line in a filediff contained in a review request is the list of review requests which affected that line in the past. The term *line* shouldn't be confused with the line in the raw diff data, rather, it refers to the line in the patched file obtained as a result of applying a line in the raw diff data.

Formally, the line change history of a line *l* in a filediff *fd* contained in a review request *rq* is defined as:

$$LCH_{fd_{rq}}(l) = (rq_1, rq_2, ..., rq_n),$$

where rq_1, rq_2 and rq_n are review requests which contain filediffs affecting line *l*. The line change history is an ordered tuple, where the first review request in the sequence, rq_1, is the latest, rq_2, the second latest and so on.

Once the line change history is found, points are assigned to the review requests within it, which is then propagated to associated users as user points. The computation of line change history and points assignment is repeated for all the lines in all the filediffs in the diff revision and the users are ranked based on their aggregated points. Finally, the top ranked users are recommended as reviewers for the given review request.

Depot revision 23		Workspace version	
1	import java.util.List;	1	import java.util.List;
2		2	
3	public class Foo {	3	public class Foo {
4	private int max;	4	private int maximum;
		5	private boolean visible;
5	private int count;	6	private long count;
6	private double avg;		
7		7	
8	public Foo() {	8	public Foo() {
	...		...

Fig. 4. Diff view of the filediff corresponding to Foo.java in Review Request R4

Review Request R3

Depot revision 20		Workspace version	
	...		...
4	private short max;	4	private int max;
	...		...

Review Request R2

Depot revision 14		Workspace version	
	...		...
3	private short m;	4	private short max;
	...		...

Review Request R1

Depot revision 10		Workspace version	
	...		...
		3	private short m;
	...		...

Fig. 5. Diff views of filediff corresponding to Foo.java in Review Requests R1, R2 and R3

1) Computing Line Change History: The filediff in a diff revision corresponds to the raw diff data generated by running the diff program on the workspace version of a source file. We used the raw diff data in auto-review to generate the user's workspace version of the source file using the patch program. The raw diff data is not easy for human reviewers to comprehend. Therefore, code review tools have support for diff viewer, which shows the depot version and workspace version of the source file side-by-side. The line change history computation can be best explained using the diff view of the filediff.

Fig. 4 shows the diff view of the filediff (in review request R4), which corresponds to depot file *Foo.java* with the depot version of 23. As shown in the diff view, lines 1,2,3,7 and 8 are unchanged in the workspace version. It also shows that line 4 and line 5 were updated, line 6 was deleted and a new line was inserted between lines 4 and 5. To trace the line change history of line 4 in the workspace version, we need to identify the review requests which contain a filediff (corresponding to *Foo.java*) that modifies or inserts this line. Assume that the review requests R3, R2 and R1 contain such a filediff. This situation is illustrated in Fig. 5. In this case, the line change history of line 4 consists of review requests R3, R2 and R1 in that order. Formally,

$$LCH_{Foo.java_{R_4}}(l_4) = (R3, R2, R1)$$

It is straight forward to compute the line change history of updated or deleted lines in the filediff. However, inserted lines require special handling, since there are no review requests in the past which affected such lines. We assume that, in most of the cases, the inserted lines are related to other lines in their proximity. This assumption is always true for lines inserted in a method. In the case of lines corresponding to new methods, it is observed that developers insert the method closer to related methods. Therefore, for inserted lines, we use the nearest existing line in the source file as a proxy for computing the line change history.

2) Reviewer Ranking: The reviewer ranking algorithm is described in Algorithm 2. The *RankReviewers* algorithm takes as input the ID of the review request and the diff revision for which reviewers are to be recommended. The algorithm returns an array of reviewers sorted in the descending order of user points.

Algorithm 2 *RankReviewers*(id, $revision$)

```
1: // id: Review Request ID, revision: Diff revision
2: ReviewRequest req ← getReviewRequest(id)
3: Diff diff ← req.getDiff(revision)
4: // Compute review request points
5: for (FileDiff fileDiff : diff.getFileDiffs()) do
6:   if (isNewFile(fileDiff)) then
7:     continue
8:   end if
9:   reqSet ← {}
10:  for (Line l : fileDiff.getLines()) do
11:    lch ← LCH_fileDiff_req(l)
12:    α ← initialPoint(fileDiff)
13:    for (ReviewRequest r : lch.history()) do
14:      r.points ← r.points + α
15:      α ← α × δ
16:      reqSet ← reqSet ∪ {r}
17:    end for
18:  end for
19: end for
20: // Propagate review request points to user points
21: userSet ← {}
22: for (ReviewRequest r : reqSet) do
23:   for (User user : r.getUsers()) do
24:     user.points ← user.points + r.points
25:     userSet ← userSet ∪ {user}
26:   end for
27: end for
28: reviewers ← Collections.toArray(userSet)
29: Sort reviewers based on points
30: return reviewers
```

The lines 2-3 retrieve the review request and the diff data within it. The **for** loop in lines 5-19 computes the line change history of each of the lines in the filediffs and assigns points to the review requests within it. The **if** statement in line 6 skips the rest of the loop for filediffs corresponding to new files, since there is no way to compute the line change history of any lines in it. The set of review requests which affected various lines in the current filediff is initialized in line 9. The **for** loop in lines 10-18 iterates over each of the lines in the current filediff, computes its line change history and assigns points to the review requests in the line change history before adding them to $reqSet$. In line 11, the line change history of the current line l is computed and assigned to lch. The function $initialPoint$ invoked in line 12 returns the initial point that is to be assigned to the first review request in lch. The function may return different initial points for different types of files. For e.g., the *.java* files in the diff can be prioritized above *.xml* files by returning a higher initial point for *.java* filediffs as compared to *.xml* filediffs. The **for** loop in lines 13-17 iterates over each of the review requests in the line change history, in the most recently affected to the least recently affected order and assigns points to them. In line 15, the point to be assigned to the next review request is reduced by a constant factor of δ, where $0 < \delta < 1$. This will ensure that the most recent review requests which affected the line are prioritized above least recent ones.

In lines 21-27, the points assigned to the review requests in $reqSet$ are propagated to user points. The users in this context refer to the submitter and the reviewers. In line 21, the set of users, $userSet$ is initialized. The **for** loop in lines 22-27 iterates over each of the review requests in $reqSet$ and assigns points to the corresponding users before adding them to $userSet$. Finally, in lines 28-30, the array of recommended reviewers ($reviewers$) is initialized using $userSet$, then sorted in the descending order of user points and returned.

3) Discussion: The RankReviewers algorithm discussed above is meant to be a replacement for the manual reviewer assignment based on file change history. As in the case of manual reviewer assignment, the RankReviewers algorithm cannot be applied to a review request with new files. At the time of writing this paper, we were working on extending RankReviewers to handle review requests with new files using syntactic similarity.

IV. Experiments

A. Automatic Review

To study the feasibility of Review Bot in the code review process, auto-reviews were generated for a set of already approved review requests and feedback was requested from a group of developers for a subset of comments in the auto-reviews.

1) Configuration: We selected 34 review requests submitted in the initial one-month period of an ongoing project for experiments. Since the review requests selected were submitted in the initial period of the project, the diff size was fairly high with an average of 20 filediffs per diff. The Java coding standard in VMware, project specific rules and best practices were used to configure Checkstyle & PMD rules. In the case of similar checks between Checkstyle and PMD, Checkstyle rules were selected. A total of 103 checks were enabled for Checkstyle and 113 for PMD. FindBugs was configured to run with the default rule set, which contains around 380 checks.

Checkstyle, unlike PMD and FindBugs outputs all source code issues with the same priority. Users should configure the priority of each of the Checkstyle rules/modules. Moreover, it uses severity levels such as *error*, *warn*, etc., instead of numerical priority levels used by PMD and FindBugs. For each of the Checkstyle modules enabled for the experiments, we manually configured its severity level and mapped the severity levels to numerical priority in the output parser.

TABLE I
AVERAGE COMMENTS PER AUTO-REVIEW [P1 = PRIORITY 1 (HIGHEST PRIORITY), P5 = PRIORITY 5 (LOWEST PRIORITY)]

Static Analyzer	Avg. comments	Avg. P1	Avg. P2	Avg. P3	Avg. P4	Avg. P5
Checkstyle	157.94	8.47	0.00	47.85	101.62	0.00
FindBugs	0.50	0.25	0.25	0.00	0.00	0.00
PMD	25.44	2.94	0.06	22.44	0.00	0.00
Sum	183.88	11.66	0.31	70.29	101.62	0.00

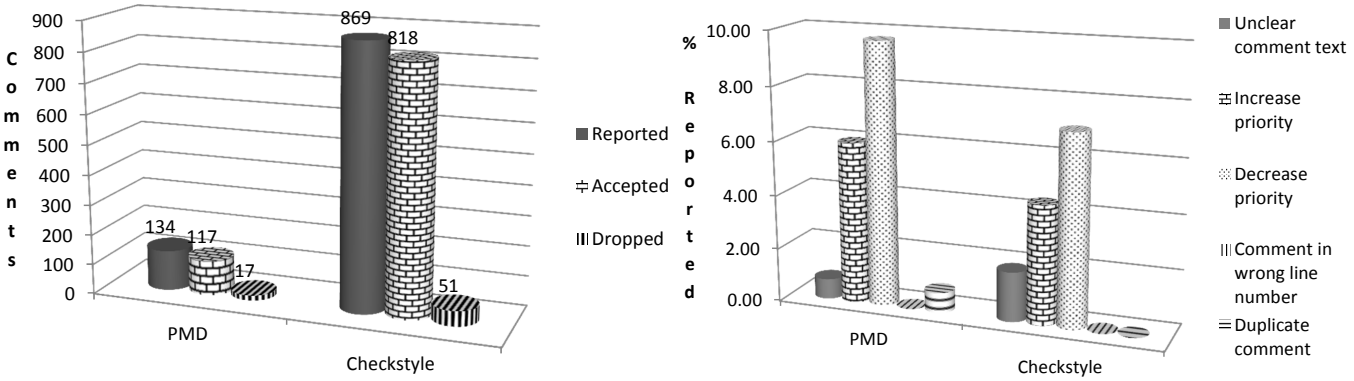

Fig. 6. User feedback on auto-reviews

2) Results: There were 6252 comments in total across 34 auto-reviews. The average comments per auto-review is listed in Table I, categorized based on the priority. Majority of the comments are Checkstyle issues with priority 3 and 4, which indicate coding standard violations and style related issues such as missing javadoc, wrong indentation, etc. The FindBugs issues are comparatively less; this could be attributed to the extensive unit testing carried out as part of the development process.

To study the feasibility of Review Bot in the code review process, a group of 7 developers working in the project under study was requested to analyze around 1000 auto-review comments and provide feedback on the following aspects:

- Whether the comment is acceptable or should it be dropped?
- Whether the comment text is clear?
- Whether the comment is a duplicate?
- Whether the priority assigned is high or low?

The developers were instructed to provide their feedback as replies to review comments using keywords. For example, if the comment text is unclear, they were asked to reply to the comment using the keyword "message". The results of the user study are shown in Fig. 6. We omit FindBugs in the results since the number of comments is too small to draw any conclusion.

As shown in Fig. 6, the developers agreed to fix more than 94% of Checkstyle comments. This might be an indication that they missed these issues before due to the lack of understanding of relevant coding standard sections or style. The fact that they agreed to fix more than 93% of all the comments reported (for all the static analyzers) is a positive indication of Review Bot's feasibility in the code review process. A manual inspection of the comments dropped revealed that the majority are style related issues without coding standard section numbers. However, there were exceptional cases where the developers consistently dropped certain issues with coding standard section numbers. One such case is the check for restricting the number of `return` statements in a method. This feedback is valuable for making appropriate changes in the coding standard.

The developers had concerns about only 14.71% of all the accepted comments, which indicates the high-quality of auto-review comments. Out of this 14.71%, 12.8% is due to improper priority values. This is not surprising since the static analyzers use pre-configured priority values for comments while the human reviewers give priority values based on the context. It is a future work to correct the priority values based on user feedback.

3) Automatic Review Screenshots: Fig. 7 illustrates screenshots of some of the comments in automatic reviews generated by Review Bot in Review Board.

B. Reviewer Recommendation

In this section, we evaluate the accuracy of the reviewer recommendation algorithm in Review Bot against another algorithm called *RevHistRECO* (Algorithm 3), which recommends reviewers based on revision history of files in the diff. The RevHistRECO algorithm is motivated by the current manual reviewer assignment.

1) RevHistRECO Algorithm: The RevHistRECO algorithm is given in Algorithm 3. As in Algorithm 2, lines 2-3 retrieves the diff for which reviewers are to be recommended. In lines 5-

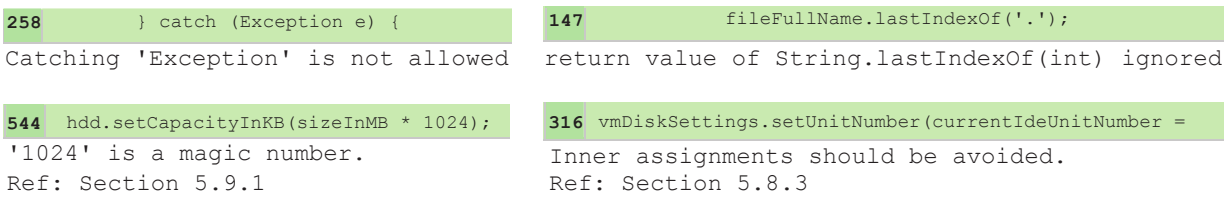

Fig. 7. Screenshots of comments in automatic reviews

Algorithm 3 *RevHistRECO*(id, $revision$)

```
1: // id: Review Request ID, revision: Diff revision
2: ReviewRequest req ← getReviewRequest(id)
3: Diff diff ← req.getDiff(revision)
4: // Find all the submitted changelists which created the base
   revision of files in this diff and assign points
5: clList ← ⟨ ⟩
6: for (FileDiff fileDiff : diff.getFileDiffs()) do
7:    if (isNewFile(fileDiff)) then
8:       continue
9:    end if
10:   cl ← findSubmittedChangelist(fileDiff)
11:   cl.points ← initialPoint(fileDiff)
12:   clList.add(cl)
13: end for
14: // Find users related to the changelists and assign points
15: userList ← ⟨ ⟩
16: for (Changelist cl : clList) do
17:    for (User user : cl.users()) do
18:       user.points ← cl.points
19:       userList.add(user)
20:    end for
21: end for
22: userSet ← consolidate(userList)
23: reviewers ← Collections.toArray(userSet)
24: Sort reviewers based on points
25: return reviewers
```

13, the list of submitted changelists in the repository ($clList$) which created the base revision of files in the given diff is computed. The *findSubmittedChangelist*(*fileDiff*) function call in line 10 queries the revision control system to read the changelist which created the base revision of the source file corresponding to *fileDiff*. In line 11, the retrieved changelist is assigned an initial point based on the source file type, as discussed in Section III-D2.

In lines 15-21, the list of users associated with $clList$ is computed and points are assigned. The users associated with a changelist include the changelist submitter and developers who approved the changelist. The $userList$ may contain duplicate entries if a particular user is associated with more than one changelist. In line 22, the $consolidate$ function finds out such entries, aggregates the points and returns a set of users. Finally, in lines 23, 24 and 25, the $userSet$ is converted to an array, then sorted based on the aggregated user points and returned.

2) Recommendation Accuracy: Even though there is no company-wide policy in choosing reviewers, most of projects in VMware require that at least one of reviewers should be an expert in the areas of the code touched in a diff. Based on this, we consider the top-k recommendation for an already approved review request to be correct, if at least one of the top-k recommended reviewers was an actual reviewer for that review request. For a dataset with N approved review requests, the top-k recommendation accuracy is defined as:

$$accuracy(k) = \frac{\text{Number of correct top-}k\text{ recommendations}}{N}$$

The recommendation algorithms were evaluated using two datasets, namely Proj_1 and Proj_2. Each of these datasets consists of approved review requests corresponding to a particular project. The Proj_1 dataset is a relatively large dataset with a total size of 7035 review requests, whereas Proj_2 consists of 1676 review requests. The number of developers who had either submitted or reviewed a review request included in the dataset is 204 for Proj_1 and 40 for Proj_2. The evaluation of Review Bot consists of iterating over the dataset in chronological order and finding recommendations based on line change history computed using all the previously submitted review requests in the project. In the case of RevHistRECO, the review requests can be considered in any order since it is not based on historical review request data.

Both the algorithms were run with $initialPoint = 1.0$ for source files (.java, .c, etc.) and 0.75 for resource files (.xml, .properties, etc.). The value of δ was set to $2/3$ for Review Bot. Table II lists the top-k recommendation accuracy of Review Bot and RevHistRECO for $k = 1$ to $k = 5$. As shown in the table, Review Bot's top-1 recommendation is accurate for both datasets in 60% of the cases. In the case of RevHistRECO, the top-1 accuracy is only 34.15% and 47.83%, respectively, for Proj_1 and Proj_2. For the relatively larger dataset Proj_1, Review Bot could achieve an accuracy of 80.85% for $k = 5$, whereas the corresponding accuracy of RevHistRECO is only 46.34%. In the case of the smaller dataset Proj_2, the top-5 accuracy of Review Bot is an impressive 92.31% compared to the corresponding RevHistRECO's accuracy of 60.39%. It is quite evident from these results that the reviewer recommendation based on line change history is far superior than the comparable method based on file revision history.

TABLE II
REVIEWER RECOMMENDATION ACCURACY

Dataset	Algorithm	Accuracy (%)				
		$k=1$	$k=2$	$k=3$	$k=4$	$k=5$
Proj$_1$	Review Bot	61.17	72.87	77.13	79.26	80.85
	RevHistRECO	34.15	41.46	43.90	45.12	46.34
Proj$_2$	Review Bot	59.92	79.35	86.23	91.50	92.31
	RevHistRECO	47.83	58.70	59.42	60.14	60.39

V. RELATED WORK

Sonar [23] is an open-source software which uses various static analysis tools for reporting source code issues. It can be integrated with popular build tools and continuous integration systems. It may be argued that the users could run Sonar and include the report generated in the review request. As mentioned in [10], the static analysis tools may receive limited use depending on how they are integrated into the software development process. We believe that the tight integration of automatic static analysis with code review, like we proposed, would be more beneficial than integrating them with the build process, since the defects can be detected and fixed much early in the development cycle.

In [24], the authors address the problem of assigning developers to bug reports. This is different from the reviewer assignment problem discussed here because review requests are not always meant for bug fixes.

VI. CONCLUSIONS AND FUTURE WORK

We proposed a tool called Review Bot based on static analysis tools to reduce the human effort in peer code reviews. Through a user study, we have shown that Review Bot can improve the review quality by finding source code issues, which need to be addressed, but could be missed during reviewer inspection. We also proposed a reviewer recommendation algorithm to ease the task of finding appropriate reviewers in a large project. Inclusion of a learning algorithm to reduce false positive comments and to correct priority values would be an interesting future work.

ACKNOWLEDGMENT

The author would like to thank all participants of the user study. He would also like to thank the anonymous reviewers for their valuable feedback.

REFERENCES

[1] I. Sommerville, *Software Engineering*. Addison Wesley, 2010.
[2] D. Huizinga and A. Kolawa, *Automated Defect Prevention: Best Practices in Software Management*. Wiley-IEEE Computer Society Press, 2007.
[3] S. C. McConnell, *Code Complete: A Practical Handbook of Software Construction*. Microsoft Press, 2004.
[4] K. Wiegers, *Peer Reviews in Software: A Practical Guide*. Addison-Wesley Professional, 2001.
[5] B. Meyer, "Design and code reviews in the age of the internet," *Commun. ACM*, vol. 51, pp. 66–71, Sep. 2008. [Online]. Available: http://doi.acm.org/10.1145/1378727.1378744
[6] P. C. Rigby and D. M. German, "A preliminary examination of code review processes in open source projects," University of Victoria, Tech. Rep. DCS-305-IR, January 2006.
[7] G. McGraw, "Automated code review tools for security," *Computer*, vol. 41, pp. 108–111, December 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1495784.1495852
[8] A. Reddy *et al.*, "Java coding style guide," *Sun MicroSystems*, 2000.
[9] G. O'Regan, *Introduction to Software Process Improvement*. Springer, 2011.
[10] N. Ayewah and W. Pugh, "The google findbugs fixit," in *Proceedings of the 19th international symposium on Software testing and analysis*, ser. ISSTA '10. New York, NY, USA: ACM, 2010, pp. 241–252. [Online]. Available: http://doi.acm.org/10.1145/1831708.1831738
[11] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner, "Thorough static analysis of device drivers," in *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, ser. EuroSys '06. New York, NY, USA: ACM, 2006, pp. 73–85. [Online]. Available: http://doi.acm.org/10.1145/1217935.1217943
[12] "Review Board," http://www.reviewboard.org/, [Online; accessed 21-Oct-2012].
[13] "Checkstyle," http://checkstyle.sourceforge.net/, [Online; accessed 21-Oct-2012].
[14] "PMD," http://pmd.sourceforge.net/, [Online; accessed 21-Oct-2012].
[15] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *SIGPLAN Not.*, vol. 39, pp. 92–106, December 2004. [Online]. Available: http://doi.acm.org/10.1145/1052883.1052895
[16] "FindBugs," http://findbugs.sourceforge.net, [Online; accessed 21-Oct-2012].
[17] N. Rutar, C. B. Almazan, and J. S. Foster, "A comparison of bug finding tools for java," in *Proceedings of the 15th International Symposium on Software Reliability Engineering*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 245–256. [Online]. Available: http://dl.acm.org/citation.cfm?id=1032654.1033833
[18] "Project Jersey," http://jersey.java.net/, 2008, [Online; accessed 21-Oct-2012].
[19] "Jackson," http://jackson.codehaus.org/, 2009, [Online; accessed 21-Oct-2012].
[20] "GemFire," http://www.gemstone.com/, [Online; accessed 21-Oct-2012].
[21] "GNU patch," www.gnu.org/s/patch/, 2009, [Online; accessed 21-Oct-2012].
[22] "Pylint - code analysis for Python," http://www.pylint.org/, [Online; accessed 17-Feb-2013].
[23] "Sonar," http://www.sonarsource.org/, [Online; accessed 21-Oct-2012].
[24] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?" in *Proceedings of the 28th international conference on Software engineering*, ser. ICSE '06. New York, NY, USA: ACM, 2006, pp. 361–370. [Online]. Available: http://doi.acm.org/10.1145/1134285.1134336

View publication stats