CFar: A Tool to Increase Communication, Productivity, and Review Quality in Collaborative Code Review

Austin Z. Henley
University of Memphis
azhenley@memphis.edu

Kıvanç Muşlu
Microsoft
kivancm@microsoft.com

Maria Christakis
MPI-SWS
maria@mpi-sws.org

Scott D. Fleming
University of Memphis
Scott.Fleming@memphis.edu

Christian Bird
Microsoft Research
cbird@microsoft.com

ABSTRACT

Collaborative code review has become an integral part of the collaborative design process in the domain of software development. However, there are well-documented challenges and limitations to collaborative code review—for instance, high-quality code reviews may require significant time and effort for the programmers, whereas faster, lower-quality reviews may miss code defects. To address these challenges, we introduce CFar, a novel tool design for extending collaborative code review systems with an automated code reviewer whose feedback is based on program-analysis technologies. To validate this design, we implemented CFar as a production-quality tool and conducted a mixed-method empirical evaluation of the tool usage at Microsoft. Through the field deployment of our tool and a laboratory study of professional programmers using the tool, we produced several key findings showing that CFar enhances communication, productivity, and review quality in human–human collaborative code review.

ACM Classification Keywords

D.2.6 Software Engineering: Programming Environments; H.5.3 Group and Organization Interfaces: Computer-supported cooperative work

Author Keywords

Programming environments; collaborative design; code review

INTRODUCTION

To "transcend the individual human mind" [8] remains a challenging and relevant problem in human–computer interaction. Central to this problem is the need for interactive systems that effectively support *collaborative design*—that is, design activities that require more knowledge, expertise, and effort than any one person can contribute [8]. Collaborating designers engage in a myriad of activities that might benefit from computer support, including brainstorming, documenting ideas, eliciting feedback, and exploring solutions. However, effective interaction designs for providing such support are often closely tied to the particulars of the domain in which a collaborative design activity takes place. Thus, researchers have investigated interaction designs for a variety of domains, including product design [46], urban design [9], interior design [42], architecture [31, 50], computer programming [52], and various forms of media creation [14, 23, 33].

One particular domain that involves extensive collaborative design is *software development*. In software development, design pervades many activities, including architectural system design, low-level code design, and software test design to name a few. Design in this domain is especially challenging, because modern software is often composed of many thousands or even millions of inter-related lines of code, requiring the work to be distributed among many programmers. Effectively designing software is further complicated by the fact that an individual programmer's design changes may have cascading effects that impact other programmers' work. To address the challenging scale and complexity of modern software, programmers attempt to coordinate their efforts. However, as a software project grows, the programmers must communicate more and more, and that communication becomes increasingly expensive due to communication overhead [18]. Thus, software development is particularly in need of systems that effectively support collaborative design.

One collaborative design activity in software development that has become particularly important is *collaborative code review*. Many software development organizations apply collaborative code review as a standard practice, including Microsoft [11], Google [35], Facebook [28], and many popular open-source software projects [51]. Following the practice, each code change must be reviewed and accepted by programmers other than the author before being merged into the product. Discussion among the programmers and multiple iterations of feedback and revision may be necessary before the change is finally accepted. A large body of evidence points to the benefits of code reviews for discovering and fixing bugs [7, 11, 26, 27, 40, 43] while also improving design aspects of the software, such as code readability and maintainability [11, 40]. Moreover, code review also has the collaborative-design benefit of helping the collaborating programmers maintain an up-to-date understanding of the evolving software design.

CHI 2018, April 21–26, 2018, Montreal, QC, Canada
© 2018 ACM. ISBN 978-1-4503-5620-6/18/04…$15.00
DOI: https://doi.org/10.1145/3173574.3173731

Although software projects commonly employ *collaborative code review systems* (e.g., Gerrit [1], Phabricator [4], and CodeFlow [11]), there remain well-documented challenges in code reviewing that limit its effectiveness. Code reviews are often time consuming, taking developers over six hours per week by one estimate [17], and developers under pressure to produce new code often struggle to find time to thoroughly review other developers' code [36]. Yet, less time spent on reviews has been shown to predict greater numbers of software bugs [29, 41, 48]. One key reason for this trend is that deeply understanding another developer's code can take substantial time and effort [11, 36, 38]. As a consequence, reviewers often find only *shallow defects*, that is, ones that are obvious or superficial [11]. For example, as a professional programmer in one study put it, "I've seen quite a few code reviews where someone commented on formatting while missing the fact that there were security issues or data model issues" [11]. Moreover, programmers, being human, have been shown to miss even such simple formatting issues in their reviews [13, 45], for example, because they have an incomplete understanding of the coding standard or simply fail to notice style violations.

To address these challenges, we designed an extension to collaborative code review systems, *CFar*, that introduces an automated code reviewer based on program-analysis technologies. In particular, our automated reviewer inserts issues detected by the analyses into a human–human collaborative code review, and in doing so, aims to achieve several key goals. One goal is to *increase communication* among the programmers—that is, to encourage human reviewers toward greater communication by freeing them from finding shallow defects and allowing them to spend more time discovering *deep defects*, that is, ones that are more challenging or involve higher-level design issues. Discussion about reviews has been shown to be a key indicator of review quality [37, 41, 48]—for example, deep understanding of code and deciding on effective solutions often requires considerable discussion [11, 38]. Another key goal is to *increase reviewer productivity*—that is, to use the automated detection of defects to reduce the overall time and effort associated with performing high-quality code reviews. A final key goal is to *reveal more defects* during code reviews—that is, to leverage the automated analyses to reduce the number of shallow defects that reviewers would otherwise miss and to facilitate human reviewers in discovering deeper issues.

To validate our design, we conducted a mixed-method empirical evaluation on-site at Microsoft. For the evaluation, we implemented our design as an extension to an existing collaborative code reviewing system, CodeFlow [11]. To integrate program analyses into CodeFlow, we leveraged existing program-analysis infrastructure provided by CloudBuild [25]. Our design implementation was production quality (as opposed to a research prototype), enabling us to study professional programmers using the tool for their actual work. In particular, our empirical evaluation had two main parts: a field deployment involving 98 professional programmers across four teams and a controlled laboratory study of seven professional programmers. As data, the field deployment provided both usage logs and survey responses, whereas the lab study provided task videos and interview responses.

The contributions of this work are as follows:

- a new tool design, CFar, for extending collaborative code review systems with an automated reviewer that uses program analyses to enhance communication, productivity, and review quality in collaborative code review, and
- the findings of a mixed-method evaluation, comprising a real-world field deployment at Microsoft and a laboratory study of professional programmers.

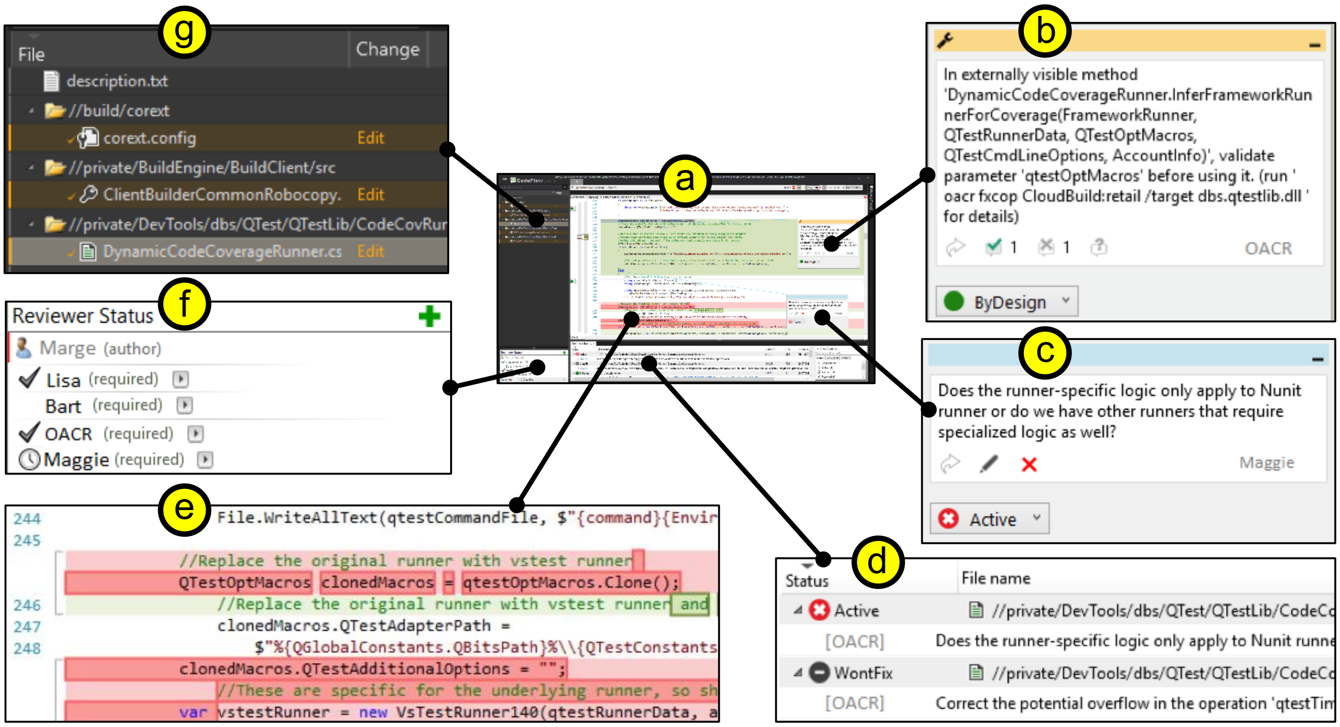
BACKGROUND: COLLABORATIVE CODE REVIEW

Collaborative code review is a quality-assurance process in which programmers wanting to add code changes to a project must submit their changes for inspection by other programmers on the team [11]. This reviewing process is notable for being lightweight, taking a relatively little amount of time, and occurring frequently throughout the development process (often multiple times per week).

The code review process is initiated by the *author*, who submits a *changeset*, the set of all the code files that have been modified, to be reviewed. Generally, the author selects the *reviewers* that have the expertise to review the changes (e.g., other programmers on the same team). Each reviewer then inspects the changes, looking for issues such as bugs, style violations, potential performance issues, and high-level design concerns. After completing their inspection, each reviewer provides this feedback to the author and must approve or reject the changeset for acceptance into the codebase. It is common for a review to take several *iterations*, consisting of reviewers providing feedback and the author making subsequent changes each time, before the changes are approved.

To support this code-review process, a number of *code review systems* with similar basic features have been created and widely adopted in practice. For example, Gerrit [1] is a particularly well-known, publicly available code review system, although many similar systems also exist (e.g., CodeFlow [11], Phabricator [4], and even GitHub and Visual Studio have such features). In this paper, we use CodeFlow as a running example since it is the system that we extended with CFar (Fig. 1).

One common feature in code review systems enables users to navigate the files in the changeset and to view differences between the previous version of the code and the proposed changes. For example, CodeFlow provides a changeset file explorer (Fig. 1g) and a code visualization that highlights lines added and removed (Fig. 1e). Another common feature enables users to attach comments to particular sections of code. For example, Fig. 1c depicts a reviewer comment, which was attached to line 236 in the code visualization. Comments may also have associated discussion threads in which the users exchange messages regarding the comments. For example, clicking the curved-arrow button in Fig. 1c would allow the user to add a discussion message to the comment. Code review systems often provide a variety of ways to explore these comments, for example, by presenting them within the code visualization or by providing a tabular listing of comments (e.g., Fig. 1d). These systems also generally provide support for managing multiple iterations, for example, by archiving and organizing previous changesets and comments. Finally,



File	Change
description.txt	
//build/corext	
corext.config	Edit
//private/BuildEngine/BuildClient/src	
ClientBuilderCommonRobocopy.	Edit
//private/DevTools/dbs/QTest/QTestLib/CodeCovRur	
DynamicCodeCoverageRunner.cs	Edit

In externally visible method 'DynamicCodeCoverageRunner.InferFrameworkRunnerForCoverage(FrameworkRunner, QTestRunnerData, QTestOptMacros, QTestCmdLineOptions, AccountInfo)', validate parameter 'qtestOptMacros' before using it. (run ' oacr fxcop CloudBuild:retail /target dbs.qtestlib.dll ' for details)

✓ 1 ✗ 1 OACR

● ByDesign

Reviewer Status

- Marge (author)
- ✓ Lisa (required)
- Bart (required)
- ✓ OACR (required)
- Maggie (required)

Does the runner-specific logic only apply to Nunit runner or do we have other runners that require specialized logic as well?

Maggie

Active

```
                File.WriteAllText(qtestCommandFile, $"{command}{Envir
        //Replace the original runner with vstest runner
        QTestOptMacros clonedMacros = qtestOptMacros.Clone();
            //Replace the original runner with vstest runner and
            clonedMacros.QTestAdapterPath =
                $"%{QGlobalConstants.QBitsPath}%\\{QTestConstants
        clonedMacros.QTestAdditionalOptions = "";
            //These are specific for the underlying runner, so sh
        var vstestRunner = new VsTestRunner140(qtestRunnerData, a
```

Status	File name
Active	//private/DevTools/dbs/QTest/QTestLib/CodeCo
[OACR]	Does the runner-specific logic only apply to Nunit runne
WontFix	//private/DevTools/dbs/QTest/QTestLib/CodeCo
[OACR]	Correct the potential overflow in the operation 'qtestTim

Figure 1. CFar-extended CodeFlow collaborative code review system. Our CFar tool extends the CodeFlow interface (a) with an automated reviewer. In particular, CFar automatically runs program analyses on the review changeset (g), and based on the results, creates comments (b) that are mapped to relevant lines in the code visualization (e). These are first-class comments very similar to ones left by human users (c). For example, they appear along with the user comments in the comment summary listing (d). The CFar reviewer ("OACR") is also included among the human reviewers in the reviewer-status listing (f), and for instance, will approve the changeset if all the CFar comments have been addressed.

these tools typically provide features for recording reviewers' statuses and decisions, for example, whether they have begun their review or whether they have accepted or rejected the changeset (e.g., Fig. 1f).

THE CFar TOOL

To increase programmer communication, enhance programmer productivity, and reveal more defects during code reviews, we designed CFar, an automated-reviewer extension to collaborative code review systems. CFar stands for CodeFlow Automated Reviewer (pronounced *see-far*). In particular, CFar introduces automated feedback into the code-reviewing process, and does so using many of the same features used by human users. Thus, in many ways, the automated reviewer appears to human users as a first-class participant in the review. A key aim of this design is to use the automated feedback to facilitate collaboration among the human users; however, the automated reviewer is not itself an intelligent conversational agent. Rather, the feedback our automated reviewer provides is based on automated program analyses. Our design was largely motivated by talking to programmers who use CFar each day, as well as the programmers who develop it.

For this work, we implemented our CFar design by extending and harnessing two industrial-strength tools, CodeFlow [11] and CloudBuild [25], respectively. CodeFlow is a collaborative code review system, similar to Gerrit, that is actively used by almost 40,000 programmers, creating more than 6300 code reviews daily. The CFar-extended version of CodeFlow incorporates our automated reviewer into the existing interface. To give our automated reviewer its program-analysis capabilities, we leveraged the CloudBuild system. CloudBuild is a cloud-based service providing resource-effective builds, tests, and program analyses. Over 4000 programmers use CloudBuild actively, requesting more than 20,000 builds daily.

In the remainder of this section, we will first describe the essential features of CFar, using our CodeFlow-based implementation as a running example, and then describe some additional details about our implementation.

Features of CFar

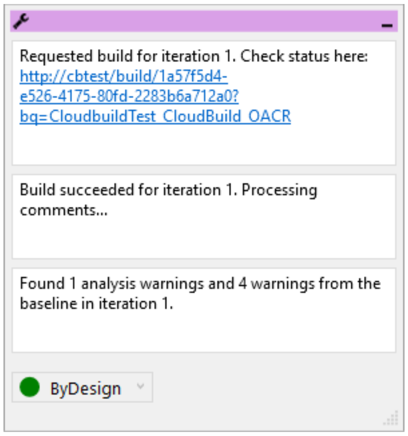
Perhaps the most central feature of CFar is that it automatically leaves review comments on the changeset code. For example, Fig. 1b depicts a comment left by the CFar reviewer. The CFar comments appear as first-class review comments in that they are inserted into the review using the same basic mechanisms as the normal human-provided comments. However, the automated reviews are visually distinct from the human ones. For example, in our CodeFlow extension, the CFar comments have a yellow border (Fig. 1b), whereas the human comments have a blue one (Fig. 1c). Moreover, the automated reviewer is listed as the reviewer who wrote the comments (e.g., the name of the program-analysis framework our tool used, "OACR", in Fig. 1b). The content of each CFar comment is produced by an automated program analysis performed by CloudBuild. To further ensure that the CFar comments appear consistent with human ones, CFar maps each comment to the relevant section of code to which the analysis output refers (e.g., the Fig. 1b comment maps to line 221 in the code visualization). In the CodeFlow code-diff visualization, each comment is connected to the relevant section of code by a thin line.

By creating CFar's analysis output as first-class comments, they are fully integrated into the normal human–human review discussion. For example, users can reply to the automated comments (e.g., by clicking the curved-arrow button in Fig. 1b), wherein they might discuss the rationale for the CFar comment, explain a proposed plan for addressing the comment, or debate the best solution for the issue. CFar comments, like user ones, also have a status ("Active", "Pending", "ByDesign", "WontFix", and "Resolved"). Although users can set the status of a CFar comment in the usual way, CFar will also change the status automatically under certain circumstances. For example, if a CFar analysis discovers that a previous comment has been fixed but the comment status is still Active or Pending, it will automatically change the status to Resolved. This feature was added after initial feedback from programmers, saying that they expected CFar to "follow up" with reviews.

The CFar comments do have one key difference with user comments: they have features for collecting feedback about the comment. In particular, each CFar comment has three buttons (depicted in Fig. 1b)—a check mark to indicate that the comment was "useful", an x mark to indicate that the comment was "not useful", and a question mark to indicate that the comment was not understandable. These buttons give users a convenient way to provide feedback on the quality of the generated comments. The system records this user feedback for use, for example, in deciding which analyses should be run or in identifying analyses that need improvement (e.g., clearer messages). For each button, the clicks collected from all users are also summed and displayed as a count next to the button, for example, to give users a convenient way to check if there is a consensus about the comment.

The CFar automated reviewer leaves or updates its comments at the start of every iteration (i.e., every round of changes). For example, when an author submits a new review (Iteration 1), CFar analyzes the codebase and changeset, and creates an initial set of comments. When a user, having made changes to the changeset, advances the code review to a new iteration, CFar will re-analyze the code, will add any new comments its analyses reveal, and will update existing CFar comments based on the changes. When dealing with large, real-world code bases, automated analyses may take a long time to complete, and CFar provides features for alerting users to the status of the automated reviewer. For example, our CodeFlow implementation displayed messages as depicted in Fig. 2. We added this feature after two programmers indicated to us that they were unsure of CFar's current status.

In addition to leaving review comments, the CFar automated reviewer also takes part in the changeset-acceptance process. For example, the CFar reviewer appears in CodeFlow's reviewer-status listing as "OACR" (Fig. 1f). If CFar detects that all its analysis comments have been addressed (i.e., neither Active nor Pending), then it will automatically accept the changeset. CFar will never reject a changeset; however, if Active or Pending comments are present, it will display its decision as pending. The goal of this feature is to make the CFar reviewer act as any other reviewer, and to follow standard practices, such as requiring there to be no rejections before accepting the code changes.



Requested build for iteration 1. Check status here: http://cbtest/build/1a57f5d4-e526-4175-80fd-2283b6a712a0?bq=CloudbuildTest_CloudBuild_OACR

Build succeeded for iteration 1. Processing comments...

Found 1 analysis warnings and 4 warnings from the baseline in iteration 1.

● ByDesign

Figure 2. Interface displaying CFar's progress on a build with automated program analyses and generating/updating analysis comments.

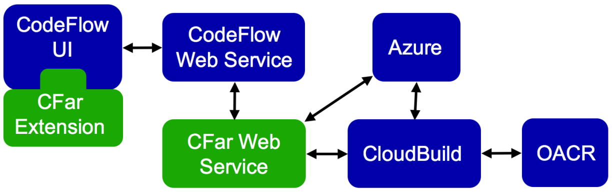


Figure 3. The architecture of CFar-extended CodeFlow. The blue boxes denote existing technologies used by our system, whereas the green boxes denote components we built.

Implementation of CodeFlow Extension

Fig. 3 illustrates the complete architecture of our CFar extension of CodeFlow. The architecture consists of two main components: (1) a front-end extension to the CodeFlow user interface, which provides the UI features described above, and (2) a back-end web service that invokes a build and processes the analysis warnings.

The CFar web service serves as the communication channel between CodeFlow and CloudBuild. Specifically, the CFar web service listens for events from the CodeFlow web service that indicate the progress of a code review. These events, for instance, indicate when a new code review is created, when a new iteration in an existing review is created, or when a review is completed. By listening to these events, the CFar web service also learns the identities of the review participants such that, for example, the automated code reviewer may be turned on only for particular groups of programmers when selectively ignoring all events about other participants.

When a new code review or iteration is created, the CFar web service first invokes CloudBuild to request a build for the new changeset (i.e., before performing any analyses). We implemented CFar to support the popular distributed version-control system Git. To perform the build for a changeset, CloudBuild needs access to a Git repository containing the code. The CFar web service provides this access by cloning the author's local repository and uploading it to Azure [3], a cloud computing service. CloudBuild can then access the repository on Azure and perform the build.

When the CFar web service requests a CloudBuild build, it also requests program analyses to be run on the (source or binary) code of each build target. To run these analyses, CloudBuild uses the OACR extensible program-analysis framework. Development teams can configure OACR to use a variety of program analyzers, such as PREfast and FxCop. For example, PREfast [5] performs intraprocedural static analysis on C and C++ source code to identify code defects, such as reliability, security, and compliance errors.

In addition to CodeFlow events, the CFar web service also subscribes to CloudBuild events that indicate the status of builds and analyses. Upon the completion of analyses, the CFar web service downloads an XML document from CloudBuild that contains all the analysis warnings returned by OACR. The CFar web service parses these warnings and collects the subset of warnings to be displayed within the code review. For example, the CFar web service filters out warnings not directly applicable to the changeset. The CFar web service then posts the warnings to the CodeFlow web service, so the warnings will appear as comments in the code review. The CFar web service also posts data about its progress to the CodeFlow service, so it can be displayed as in Fig. 2.

METHOD

To evaluate the extent to which CFar achieves its goals, we conducted a mixed-method empirical evaluation at Microsoft. In research contexts with high variability, such as software engineering, all empirical methods have limitations—for example, laboratory experiments generally emphasize control at the expense of realism, whereas case studies make the opposite trade-off. A *mixed-method* approach to investigation aims to address this issue by applying multiple study designs to the same research questions with the idea that "the weaknesses of one method can be compensated for by the strengths of other methods" [22].

The research questions that we addressed with our mixed-method empirical evaluation of CFar were as follows:

RQ1: Did CFar increase communication among programmers?
RQ2: Did CFar increase productivity of programmers?
RQ3: Did CFar improve code quality?
RQ4: Did programmers like CFar?

Our mixed-method approach to addressing these questions comprised two studies: a real-world field deployment of the tool at Microsoft and a laboratory study of professional programmers using the tool. For the field deployment, we sacrificed control and the ability to observe details of how programmers used the tool, in exchange for a realistic evaluation context in which real-world programmers were using the tool on their actual development tasks. In contrast, for the lab study, we sacrificed realism, constraining task time, codebase size, and the tasks to be performed, in exchange for the ability to control the participants' tasks, thus facilitating direct comparisons, and to directly observe participants as they worked with CFar. In the remainder of this section, we detail the methods we used for each of these studies.

Field Deployment

We deployed CFar to 98 programmers across three teams at Microsoft to be used in the programmers' code reviews for 15 weeks. In the remainder of the paper, we refer to these participants as FP-*X*, where FP refers to Field Participant and *X* refers to the participant number (e.g., FP-15 refers to field-study participant #15). We selected these teams because they use all of the necessary components for CFar (CodeFlow, CloudBuild, and OACR), and thus already employ a workflow of performing code reviews. These teams are located in the US, with most members co-located in the same offices, and are composed of predominately male programmers with a large range of professional experience. Two of the teams work on customer-facing products while the third team works on internal productivity services. Additionally, a fourth team of programmers used the front-end of CFar with their own analysis and logging capability as back-end. We sent an email to the programmers explaining how CFar works along with examples of what it would add to their reviews. Using the tool was voluntary and could be enabled or disabled for each programmer and review. The specific program analyses used by CFar were entirely up to the individual teams. Thus, we did not modify their existing analysis configurations (e.g., one team could have been running a security analysis while another style and concurrency analyses).

By the end of the field deployment, CFar had been used for 354 code reviews that were created by 41 unique review authors and included 883 unique reviewers. Of those reviews, 30 reviews contained at least one analysis comment, with a total of 149 analysis comments overall. The number of analysis comments might have been greater, but the teams studied had been using the program analyzers in their existing workflow for some time and, for example, had already configured the tools to suppress many warnings.

Following the deployment, we emailed the 98 programmers from the three teams to take part in a survey[1], so we could better understand their usage and opinions of CFar. The questions asked whether CFar decreased or increased communication, productivity, and code quality. Most questions had Likert-scale responses along with a textbox to provide further explanation. The survey was anonymous—respondent anonymity has been shown to increase response rates [49] and leads to more-candid responses. We also ensured that the survey took only about 10 minutes, as long surveys may deter participation. 33 of the programmers filled out the survey, yielding a 33.7% response rate, which is considered relatively high [44]. One possible reason for our high response rate is that we targeted only actual users of our tool as opposed to "cold emailing" individuals with no prior knowledge of the research. During analysis, we excluded one respondent's replies because they indicated that CFar was not enabled on their code reviews.

[1] https://github.com/human-se/cfar-survey

Lab Study

To gain rich qualitative insights about CFar, we ran a laboratory user study of programmers performing two code reviews, one with and one without our tool. This allowed us to closely observe programmers as they performed code reviews and receive feedback from them regarding our tool.

We emailed 36 programmers from one of the three teams to which we had deployed our tool. This team was selected due to their close proximity to us (within walking distance). These programmers already had experience with CodeFlow, CloudBuild, and OACR. Of them, seven programmers (six male, one female) took part in the lab study. We refer to these participants as LP-*X*, where LP refers to Lab Participant and *X* refers to the participant number (e.g., LP-3 refers to lab-study participant #3). All participants held bachelor's degrees. Additionally, three also held a master's degree and one a PhD degree. On average, the participants had 15 years of programming experience ($SD = 10$). They reported performing an average of 25 code reviews per week ($SD = 44.8$) and requesting an average of three reviews per week ($SD = 1.6$).

The code we asked participants to review was from an actual project at their company and from commits that had been previously made by other programmers. We selected code from a project with which all of our participants would be familiar. We ensured that none of the participants had previously reviewed the code or committed changes to it by checking the commit logs as well as by asking them at the start of the lab session. Our goal was to enhance validity of our study by choosing actual commits to review and by ensuring the code was not completely foreign to the participants.

The participants took part in individual lab sessions lasting approximately one hour. First, participants were read a summary of the study and then filled out a background questionnaire. Next, they were asked to complete two code reviews, with 25 minutes to complete each. Analysis comments were added to one of the reviews, but not the other. The order of the reviews was the same for all participants but which review received the analysis comments was chosen randomly. If participants did not complete a review within 25 minutes, they were asked to stop. To better understand the participants' behaviors, we asked them to "think aloud" [24] as they performed the reviewing tasks. After completing the two code reviews, each participant took part in a semi-structured interview regarding their thoughts on the CFar tool. As data, we collected audio and screen-capture video of each participant's session.

RESULTS

To address our research questions, we analyzed the usage logs and survey responses from the field deployment, and the task videos and interview responses from the lab study. We now report the results of our analyses for each research question.

RQ1 Results: Increased Communication

As Fig. 4 shows (left bar), over half of the survey respondents (61%) reported that CFar enhanced their team's collaboration. Moreover, as the same figure shows (right bar), nearly half of the programmers (45%) reported that CFar inspired more conversations. In contrast, only a handful of respondents indicated that CFar either had no effect or reduced collaboration and that it reduced communication (14% and 7%, respectively).

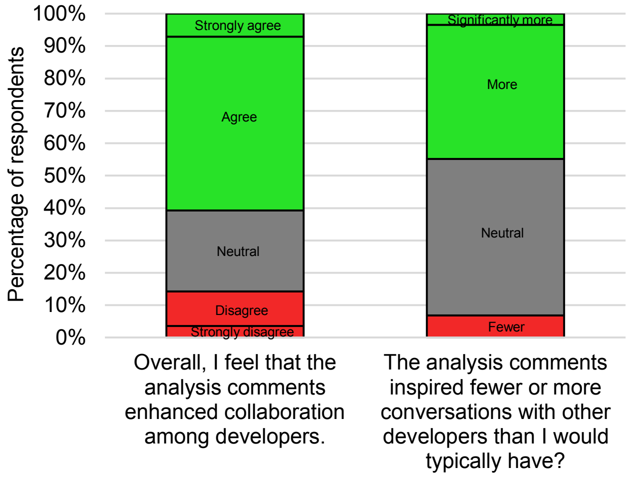


Figure 4. Field-deployment survey responses regarding the impact of CFar on programmer communication during code reviews (RQ1). The green portion of each bar denotes responses that support CFar's effectiveness; red denotes responses against; and gray denotes neutral responses.

In the field-deployment survey, we also asked programmers an open-ended question about which analysis comments they discuss with other programmers. 40% of the respondents said that they communicate with other programmers about *all* of the analysis comments. However, during the field deployment, participants wrote a reply to only 9% of the analysis comments. Although these results at first seem contradictory, they may be explained by the fact that each team in our study worked in an open-plan office, and tended to engage in in-person discussions (as opposed to using CodeFlow). For example, LP-4 described situations in which he will go talk to the review author before even looking at a review:

> LP-4: "If the change is anything above a minor bug fix... if it typically touches more than 5 or 10 files or has some kind of design thing, I go talk to them."

As Fig. 5 shows, CFar inspired conversation topics that ranged from shallow defects to deep issues. For example, the Coding Style category would have tended toward shallow defects. In contrast, the Refactoring, Code Smells, and High-Level Design categories would have tended toward deeper design issues. The Implementation Details category likely contained a mix of shallow defects (e.g., minor bugs) and deeper issues (e.g., subtle security vulnerabilities and concurrency errors).

Programmers also made statements in their open-ended responses that further indicate that CFar led to additional conversations. As one programmer explained:

> FP-14: "Having some comments helped start the conversations that might be missed until last minute, so their addition is a net positive."

Another programmer recounted a situation in which CFar caused him to talk to a fellow programmer:

> FP-33: "I followed up with the tool owner to understand whether there was a performance impact or not. I probably would not have if it hadn't been called out in the review."

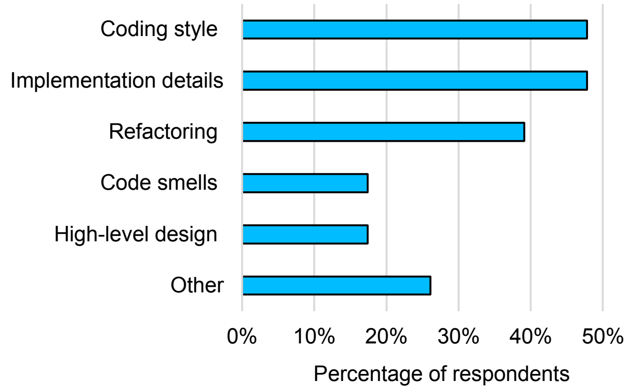


Figure 5. Field-deployment survey results indicating the programmer-conversation topics inspired by CFar during code reviews. The Other category consisted of "unit tests", "performance issues", "educating junior programmers", and "build-breaking issues" (none received more than two responses).

Additionally, two more programmers provided their thoughts on why the comments enhanced communication:

> FP-6: "Different programmers have different perspectives about the same problem or issue or solution. It's beneficial to talk with others and acquire inspiration."
>
> FP-13: "The auto-generated code review comments have prompted me to ask questions of fellow programmers about how best to resolve the problem at a point in development where it still feels productive to do the right thing rather than the expedient thing. As a result, the discussions I've had have been more in depth and just more usefully focused on what the right solution is."

RQ2 Results: Increased Productivity

As Fig. 6 shows (left bar), a considerable number of the survey respondents (38%) indicated that CFar increased their productivity. One reason reported by programmers was that it largely freed them from having to provide feedback about shallow bugs. A number of programmers remarked on this fact:

> FP-33: "Anything that can be automated should be. Reviewers should focus on critical thinking and bots should do the mindless clerical work."
>
> LP-7: "I wouldn't even have to look for those things. I would just look for higher-level things."
>
> LP-1: "When you are thinking the same thing that [the tool] has already pointed out, you don't need to focus so much on that part anymore. I also learn about things I didn't see in the review."

Interestingly, these remarks also suggest an interrelationship between productivity and code quality. In particular, by freeing the programmers from dealing with shallow defects, they were able to invest more effort into finding and discussing deep defects. One programmer commented on the importance of this benefit:

> LP-4: "Maybe earlier in my career I would have gone right into the code and found logical issues and fine grained stuff. But those are not as interesting. I want to provide design feedback."

Another productivity benefit that programmers cited was that CFar delivered feedback quickly and early in the code-review process. For example, one programmer elaborated on this productivity benefit:

> FP-6: "The instant feedback—I didn't have to wait for anyone's comments."

Several others explained that, even though they were already using program-analysis tools in later parts of their workflow, CFar helped them save time by delivering the analysis feedback sooner:

> FP-24: "Saves me time to fix the same issue [now, rather than] very late."
>
> FP-13: "I'm glad to have had the warnings pulled earlier in my development loop so I can address them earlier."
>
> FP-21: "Many comments are things that I would have to fix anyway, so I like knowing about those things sooner (in Code Review) rather than later (when I'm trying to check-in or after). Having CodeFlow automatically give me extra information and/or an early heads-up about things to fix is a good time saver."

In contrast to the productivity benefits expressed by these programmers, several (19%) indicated that CFar decreased their productivity. From the lab study, several participants expressed that the analysis comments cause them to want to view relevant code that is not included in the code review, and so they must open a code editor to view it (CodeFlow does not currently support this). Another reason cited for CFar having a negative productivity impact was an overload of CFar comments:

> FP-29: "The comments simply get in the way of someone trying to do a review."
>
> FP-28: "Too much noise."

RQ3 Results: Improved Code Quality

As Fig. 6 shows (right bar), nearly half of the survey respondents (48%) indicated that CFar helped increase the quality of code. Moreover, only one programmer responded that the use of CFar decreased code quality.

During the field deployment, CFar posted a variety of analysis warnings as comments in code reviews, summarized in Fig. 7. All warnings concerned shallow defects, and in particular, program behavior, code style, and API usage. For example, the most common warning is about parameters that have not been validated or null checked. Other popular warnings include those regarding API usage. Four of the warning types shown in Fig. 7 are about using suitable XML or string libraries.

One reason why these CFar comments improved code quality is that programmers did not simply ignore them, but rather, acted on them. Recall that each review comment in our CFar-extended CodeFlow has a status (i.e., Active, Resolved, By-Design, etc.). At the end of the field deployment, only 3% of the analysis comments remained Active (i.e., unaddressed) in the completed code reviews, whereas 97% had their statuses changed either by a programmer or by CFar itself, which automatically sets the status of an analysis comment to Resolved if the corresponding code issue has been fixed. Furthermore, in the field-deployment survey, 33% of the respondents said that they responded to all analysis comments no matter what code issue they concern.

In addition to responding to CFar comments, the programmers also indicated that they understood the CFar comments. Recall that each CFar comment has buttons (useful, not useful, or do not understand) that users could click to provide feedback on the comment. During the field deployment of CFar to the fourth programmer team, which used the front-end component of our architecture, we collected this user feedback.

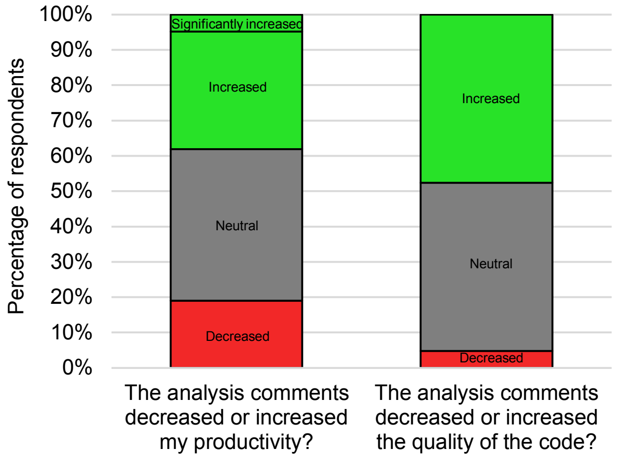


Figure 6. Field-deployment survey results regarding the impact of CFar on programmer productivity (RQ2) and code quality (RQ3). The green portion of each bar denotes responses that support CFar's effectiveness; red denotes responses against; and gray denotes neutral responses.

There were 119 votes in total, of which 74 were useful and 45 not useful. No programmer indicated not understanding a comment. This is particularly encouraging given that prior research has found that one of the biggest barriers to adopting program analysis is the difficulty to understand the generated warnings [21]. Displaying the warnings in CFar as comments placed directly on top of the relevant code may have aided programmer understanding and avoided this common barrier.

Many of the programmers offered explanations in the survey as to how CFar assisted them in finding and addressing code issues. For example, one programmer provided his rationale as to why CFar helped him:

> LP-6: "...in code reviews you can only do so much. You cannot really go through all the details and go through all the dots to find the bugs. These kinds of comments make it really useful."

Several other participants brought up that CFar uncovered issues that a human reviewer would not have:

> FP-24: "Some errors are just too hard for a human to notice."
>
> LP-5: "These are often things people don't catch but are supposed to in code reviews or things you would make a comment on anyway."
>
> FP-18: "Warnings make me think of something that I otherwise wouldn't have."
>
> FP-30: "Things I might have missed earlier would be pointed out, and I'd go look at that piece of code in more detail."

While there was a generally strong consensus that CFar enhanced code quality, some programmers provided neutral or negative responses. A potential reason for the high number of neutral responses is that programmers were required to change their workflow in order to utilize CFar's features. Those that did overcome this barrier expressed issues with the program analyses used by CFar:

> FP-13: "So far I've only seen warnings about potential hazards that turned out not to be problems so code quality has been unaffected."
>
> FP-32: "I dislike the comments I see because they're all redundant. . . "

RQ4 Results: Found Useful

As Fig. 8 shows (left bar), all but one of the field-deployment survey respondents indicated that they found the CFar comments useful. Moreover, the figure (right bar) shows that a strong majority of respondents liked the CFar comments (only one participant did not). These sentiments were echoed by the lab-study participants: all seven participants in the lab study indicated that they found the analysis comments useful and held a favorable opinion of CFar.

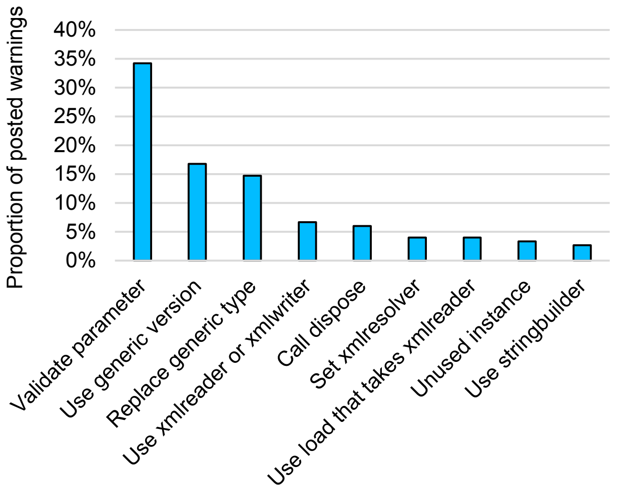


Figure 7. Field-deployment log results regarding the types of analysis warnings that CFar posted as review comments. The chart shows only the top-ten most commonly-posted warnings.

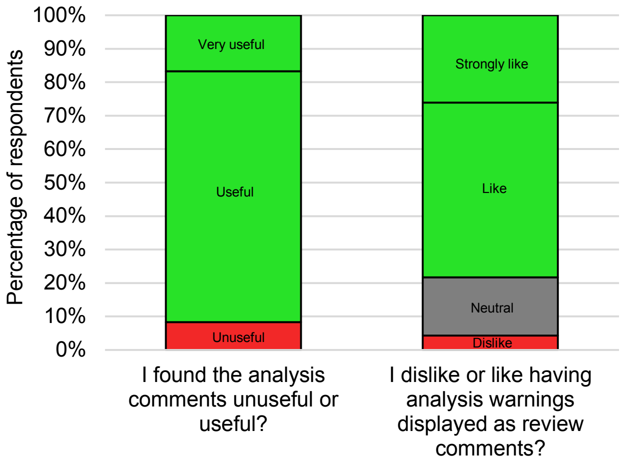


Figure 8. Field-deployment survey results regarding the usefulness of the CFar tool. The green portion of each bar denotes responses that support CFar's effectiveness; red denotes responses against; and gray denotes neutral responses.

Beyond the programmer opinions addressed in our other research questions, our participants also provided feedback on a variety of additional aspects that they liked about CFar. Two participants made comments showing their general enthusiasm for the tool's potential:

> FP-13: "It seems like the auto-generated comments I've seen are the tip of the iceberg... I'm optimistic that there's a lot of untapped potential."
>
> LP-3: "I think this is an awesome space that I would love to see even more static analysis done to make my job of code reviewing even easier."

Another participant reported a great example of when CFar can help educate programmers:

> FP-22: “Currently, I have seen [CFar] comment on certain API usages. This has led to some wonderful learning opportunities that would not have surfaced otherwise.”

Although the majority of participants indicated that CFar was useful, a couple participants contradicted the others. In particular, they tended to be critical of the CFar comments:

> FP-32: “Too many useless comments.”
>
> FP-11: “The comments I have seen have been too trivial.”

DISCUSSION

Overall, the results of our empirical evaluation of CFar were favorable. A considerable proportion of the programmers reported that CFar increased their communication and enhanced collaboration during code reviews (RQ1 results). Furthermore, many programmers reported that CFar increased their productivity, with a key reason being that it freed them from dealing with many shallow defects (RQ2 results). In addition to productivity gains, many also reported that CFar helped increase the quality of their code, largely because it caught issues a human would miss (RQ3 results). Lastly, nearly all programmers reported finding CFar useful, and a majority indicated that they liked having the automated code reviewer (RQ4 results).

However, in addition to this strongly favorable feedback, some programmers also pointed out limitations of the CFar tool. In particular, their comments suggested opportunities for improving CFar with respect to the quantity and quality of feedback produced. Here, we discuss these limitations in more detail along with promising approaches for addressing them.

Reducing Information Overload in CFar

One limitation of CFar revealed by our results was that several programmers reported feeling overwhelmed by the size and/or number of CFar comments. For instance, several programmers indicated that this issue hindered their productivity (RQ2 results) and overall perception of the usefulness of CFar (RQ4 results). Additionally, one of the field-deployment survey questions asked for the programmers’ opinions about the quantity of CFar comments. The programmers were somewhat divided in their opinions on this question. On the one hand, 31% of the programmers indicated that there were too few comments, and several of them indicated that they would like to see more comments from different types of program analyses. However, on the other hand, 19% of programmers reported that there were too many CFar comments, supporting the idea that information overload was a problem.

One possible approach for addressing the information-overload problem in CFar is to provide additional support for efficiently eliding CFar comments using filtering criteria based on characteristics of the program analyses. In our study, each development team could enable and disable program analyzers using OACR; however, using these features was tedious and time consuming, for example, because to update the CFar comments required rerunning the entire build and analysis. It stands to reason that more-usable program-analysis filtering and eliding features within the code-review system could address this problem. Indeed, several programmers expressed wanting greater support for filtering CFar comments:

> FP-28: “Must be able to turn off comments by category / type.”
>
> FP-14: “If we started to have more detailed analysis in CodeFlow, I’d like to see it able to be filtered out so human comments can be given priority. Too many comments would definitely detract from use and usefulness of the tool.”
>
> FP-32: Suggested “static code analysis vs. not code analysis” as types of filters to add to CFar.

As it happens, CodeFlow already provides comment-filtering features (e.g., for filtering by author), and it would be relatively straightforward to extend those features with filters specific to program analyses.

In addition to reducing the number of warnings to alleviate information overload, the individual warnings themselves could be reduced in size. In particular, their textual content could be shortened or elided. The warnings from the program-analysis back-end used by our tool were originally designed to be displayed in a long list, much like compiler warnings; however, CFar fundamentally changes the context in which the warnings are displayed by annotating code with them. A prior study using eye-tracking also observed programmers having difficulties reading automated analysis warnings [16], and other researchers proposed visualizations to improve comprehension [15]. Such approaches could be applied to our CFar tool to further help address the problem of information overload.

Improving CFar Comment Relevance

Several programmers expressed concern about how the program analyses used by the automated code reviewer need to be configured correctly for each team. Our tool already uses a team’s current configuration for OACR; however, the programmers may be more concerned now that the analysis warnings are displayed in their code reviews, as opposed to a log file that is easy to ignore. In fact, the programmers of one of the teams we studied had ignored the OACR log file for six months until we deployed the automated code reviewer to them. Several programmers also provided feedback, voicing their concerns as to how important proper configuration is:

> FP-14: “However each team or programmer may have different styles, so it would have to be only the most important things to highlight or people would ignore them or stop using [the tool] as it would get in the way.”
>
> LP-2: “There are a lot of rules I don’t ever want turned on. They are just too chatty. I’d be like, ‘that’s a stupid rule; let’s turn that off.’ There’s a ton of bad ones. There’s a ton of good ones too.”
>
> LP-7: “The question is will [the static analyses] be used right by the team... It’s a function of, even if OACR has it, will they enable it or do the work to keep it clean?”

One possible way to configure an automated code reviewer for the ever-changing needs of a team or project is to leverage feedback from the programmers themselves. Such data may be collected by checking the status of analysis comments (e.g., Resolved versus WontFix) or by counting the CFar comment feedback (recall from Fig. 1b; useful, not useful, or do not understand). If these data indicate that a certain type of analysis comment is consistently not fixed or disliked by programmers, then comments of this type could be excluded from future code reviews of the particular team. Other researchers have proposed similar ideas, such as Google’s Tricorder tool [45], which removes analyses that are deemed unhelpful; however, Tricorder does so globally rather than per team.

RELATED WORK

Similar to our work, several prior researchers have explored integrating program analyses into code-reviewing tools. For instance, researchers at Google integrated warnings from the FindBugs static analyzer into their Mondrian code-review system [10]. However, this effort ran into scaling problems, which caused stale or delayed analysis warnings, and produced many automated comments that were never addressed by programmers [45]. To address these issues, researchers at Google built Tricorder [6, 45], a program analysis ecosystem that scales and has features that enable programmers to write custom analyzers and that detect ineffective analyzers and remove them from the system. Tricorder analysis warnings are posted in Google's internal code-reviewing tool, which has features similar to CodeFlow [11]. Like our CFar tool, Tricorder has been empirically shown to help reveal code defects [45]; however, unlike CFar, Tricorder's creators have made no claims about (and thus not empirically evaluated) Tricorder's influence on programmer communication and productivity in code reviews.

Similar to Google's efforts, VMware also proposed a tool called Review Bot [12, 13] for integrating program analysis with code reviewing. A preliminary empirical evaluation of Review Bot demonstrated the tool's potential for finding defects—the programmers stated that they would fix 93% of Review Bot's automatically generated review comments. However, similar to the work at Google, the claims and evaluation did not specifically address the issues of programmer communication and productivity during code reviews.

In addition to the Google and VMware tools, several other code-review tools have been proposed that incorporate program analyses; however, these other tools either have no reported evaluations or the results were negative. One of the earliest code-review tools that incorporated static analyses was NASA JPL's SCRUB [34]. More recently at Facebook, a program analyzer, Infer [20, 19, 2], was incorporated into the collaborative code review tool, Phabricator [4]. Lastly, Octopull [32] incorporated static program analyses into the GitHub platform's interface. However, no empirical evaluations have been published for SCRUB and Phabricator/Infer, and in an evaluation involving CS undergraduate students, Octopull showed no significant effect [32].

THREATS TO VALIDITY

Every empirical study has threats to validity [30], and we applied a mixed-method approach to reduce some of the threats to our investigation. For both our studies, a key threat to external validity [47] was that we studied programmers and tools from a single software organization; thus, our findings may not generalize to other software organizations. Furthermore, most of our participants were male, which may prevent our findings from generalizing. Reactivity effects also posed a threat to external validity in both our studies. That is, the programmers may have acted differently than they normally would, because they knew they were being observed or guessed that the researchers had created CFar. This may have occurred in our survey, since most of the open-ended responses were positive.

Our two studies each had their own key threats to validity as well. The use of a Likert-style survey in the field deployment created a threat to construct validity [39], and to help mitigate this threat, we asked respondents to provide open-ended explanations to clarify each of their answers. The use of controlled tasks and a controlled working environment in the lab study created a threat to ecological validity. We sought to offset this threat by triangulating the results with the field deployment in which participants used the tool on their everyday work in a real-world setting. Moreover, we further sought to mitigate the threat in the user study by using code that was actually submitted for review at the company and that was taken from a project familiar to all of the participants.

CONCLUSION

In this paper, we introduced the novel CFar tool design for extending collaborative code review systems with an automated code reviewer that uses program analyses to enhance communication, productivity, and review quality in human–human collaborative code review. Our mixed-method empirical evaluation of CFar produced several key findings:

- RQ1 (communication): 45% of programmers who used CFar for their work indicated that the tool increased communication, and over 60% indicated that it enhanced collaboration. (Only 7% and 14% of participants, respectively, disagreed with these benefits.)
- RQ2 (productivity): 38% of programmers who used CFar indicated that it increased their productivity (versus only 19% who disagreed), and multiple participants indicated that a key reason was that it freed them from dealing with shallow defects.
- RQ3 (code quality): 48% of programmers who used CFar indicated that it helped increase code quality (versus one who disagreed), and several expressed that a key reason was that it identified defects that a human reviewer would miss.
- RQ4 (user opinions): All but one programmer who used CFar found it useful, and 69% expressed that they liked the tool (versus only one programmer who did not).

We hope that CFar and our findings represent a substantial step toward more efficient and more effective collaborative code review systems. A promising direction for future work is to explore novel ways in which a human may interact with an automated code reviewer. For instance, to debug an analysis comment, a review participant might require more information than that provided by the analysis warning. In such cases, the automated reviewer could display, upon request, the code parts that are to blame for the warning and even suggested fixes. Our tool design and implementation elicited considerable optimism from programmers at Microsoft, which revealed an even larger opportunity for tools to improve code reviewing. Tools like CFar could continue to help programmers, as was the case for one of our participants:

> LP-4: "I can use my time and energy in some other place of the code that is more important."

Acknowledgment

This material is based upon work supported by Microsoft and the National Science Foundation (NSF) under Grant No. 1302117.

REFERENCES

1. Accessed Jan 2018. *Gerrit*. `https://www.gerritcodereview.com/`.
2. Accessed Jan 2018. *Infer*. `http://fbinfer.com/`.
3. Accessed Jan 2018. *Microsoft Azure*. `https://azure.microsoft.com`.
4. Accessed Jan 2018. *Phabricator*. `https://www.phacility.com/`.
5. Accessed Jan 2018. *PREfast*. `https://msdn.microsoft.com/en-us/library/ms933794.aspx`.
6. Accessed Jan 2018. *Tricorder*. `https://github.com/google/shipshape`.
7. A. Frank Ackerman, Lynne S. Buchwald, and Frank H. Lewski. 1989. Software Inspections: An Effective Verification Process. *IEEE Softw.* 6, 3 (May 1989), 31–36.
8. Ernesto Arias, Hal Eden, Gerhard Fischer, Andrew Gorman, and Eric Scharff. 2000. Transcending the Individual Human Mind—Creating Shared Understanding Through Collaborative Design. *ACM Trans. Comput.-Hum. Interact.* 7, 1 (March 2000), 84–113.
9. Ernesto Arias, Hal Eden, and Gerhard Fisher. 1997. Enhancing Communication, Facilitating Shared Understanding, and Creating Better Artifacts by Integrating Physical and Computational Media for Design. In *Proceedings of the 2nd Conference on Designing Interactive Systems: Processes, Practices, Methods, and Techniques (DIS '97)*. ACM, 1–12.
10. Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, John Penix, and William Pugh. 2008. Using Static Analysis to Find Bugs. *IEEE Softw.* 25, 5 (Aug. 2008), 22–29.
11. Alberto Bacchelli and Christian Bird. 2013. Expectations, Outcomes, and Challenges of Modern Code Review. In *Proceedings of the 35th International Conference on Software Engineering (ICSE '13)*. IEEE Computer Society, 712–721.
12. Vipin Balachandran. 2013a. Fix-it: An Extensible Code Auto-Fix Component in Review Bot. In *Proceedings of the 13th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM '13)*. IEEE Computer Society, 167–172.
13. Vipin Balachandran. 2013b. Reducing Human Effort and Improving Quality in Peer Code Reviews Using Automatic Static Analysis and Reviewer Recommendation. In *Proceedings of the 35th International Conference on Software Engineering (ICSE '13)*. IEEE Computer Society, 931–940.
14. Patti Bao, Elizabeth Gerber, Darren Gergle, and David Hoffman. 2010. Momentum: Getting and Staying on Topic During a Brainstorm. In *Proceedings of the 28th International Conference on Human Factors in Computing Systems (CHI '10)*. ACM, 1233–1236.
15. Titus Barik, Kevin Lubick, Samuel Christie, and Emerson R. Murphy-Hill. 2014. How Developers Visualize Compiler Messages: A Foundational Approach to Notification Construction. In *Proceedings of the 2nd IEEE Working Conference on Software Visualization (VISSOFT '14)*. IEEE Computer Society, 87–96.
16. Titus Barik, Justin Smith, Kevin Lubick, Elisabeth Holmes, Jing Feng, Emerson Murphy-Hill, and Chris Parin. 2017. Do Developers Read Compiler Error Messages?. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*. ACM, 575–585.
17. Amiangshu Bosu and Jeffrey C. Carver. 2013. Impact of Peer Code Review on Peer Impression Formation: A Survey. In *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '13)*. IEEE Computer Society, 133–142.
18. Frederick P. Brooks, Jr. 1975. *The Mythical Man-Month* (1st ed.). Addison-Wesley.
19. Cristiano Calcagno and Dino Distefano. 2011. Infer: An Automatic Program Verifier for Memory Safety of C Programs. In *Proceedings of the 3rd International Symposium on NASA Formal Methods (NFM '11 LNCS)*, Vol. 6617. Springer, 459–465.
20. Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter W. O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. 2015. Moving Fast with Software Verification. In *Proceedings of the 7th International Symposium on NASA Formal Methods (NFM '15 LNCS)*, Vol. 9058. Springer, 3–11.
21. Maria Christakis and Christian Bird. 2016. What Developers Want and Need from Program Analysis: An Empirical Study. In *Proceedings of the 31st IEEE / ACM International Conference on Automated Software Engineering (ASE '16)*. ACM, 332–343.
22. Steve Easterbrook, Janice Singer, Margaret-Anne Storey, and Daniela Damian. 2008. Selecting Empirical Methods for Software Engineering Research. In *Guide to Advanced Empirical Software Engineering*. Springer, Chapter 11, 285–311.
23. W. Keith Edwards and Elizabeth D. Mynatt. 1997. Timewarp: Techniques for Autonomous Collaboration. In *Proceedings of the 15th International Conference on Human Factors in Computing Systems (CHI '97)*. ACM, 218–225.
24. K. Anders Ericsson and Herbert A. Simon. May 1980. Verbal Reports as Data. *Psychological review* 87, 3 (May 1980), 215–251.
25. Hamed Esfahani, Jonas Fietz, Qi Ke, Alexei Kolomiets, Erica Lan, Erik Mavrinac, Wolfram Schulte, Newton Sanches, and Srikanth Kandula. 2016. CloudBuild: Microsoft's Distributed and Caching Build Service. In *Proceedings of the 38th International Conference on Software Engineering—Companion Volume*. ACM, 11–20.

26. Michael E. Fagan. 1976. Design and Code Inspections to Reduce Errors in Program Development. *IBM Syst. J.* 15, 3 (Sept. 1976), 182–211.

27. Michael E. Fagan. 1986. Advances in Software Inspections. *IEEE Trans. Softw. Eng.* 12, 1 (Jan. 1986), 744–751.

28. Dror Feitelson, Eitan Frachtenberg, and Kent Beck. 2013. Development and Deployment at Facebook. *IEEE Internet Computing* 17, 4 (July 2013), 8–17.

29. Andre L. Ferreira, Ricardo J. Machado, Jose G. Silva, Rui F. Batista, Lino Costa, and Mark C. Paulk. 2010. An Approach to Improving Software Inspections Performance. In *Proceedings of the 26th IEEE International Conference on Software Maintenance (ICSM '10)*. IEEE Computer Society, 1–8.

30. Nahid Golafshani. 2003. Understanding Reliability and Validity in Qualitative Research. *The Qualitative Report* 8, 4 (Dec. 2003), 597–606.

31. Tovi Grossman and Ravin Balakrishnan. 2008. Collaborative Interaction with Volumetric Displays. In *Proceedings of the 26th International Conference on Human Factors in Computing Systems (CHI '08)*. ACM, 383–392.

32. Reinier M. Hartog. 2015. *Octopull: Integrating Static Analysis with Code Reviews*. Master's thesis. Delft University of Technology.

33. Otmar Hilliges, Lucia Terrenghi, Sebastian Boring, David Kim, Hendrik Richter, and Andreas Butz. 2007. Designing for Collaborative Creative Problem Solving. In *Proceedings of the 6th Conference on Creativity and Cognition (C&C '07)*. ACM, 137–146.

34. Gerard J. Holzmann. 2010. SCRUB: A Tool for Code Reviews. *Innov. Syst. Softw. Eng.* 6, 4 (Dec. 2010), 311–318.

35. Niall Kennedy. 2006. Google Mondrian: Web-based Code Review and Storage. (2006). Accessed Nov 2017.

36. Oleksii Kononenko, Olga Baysal, and Michael W. Godfrey. 2016. Code Review Quality: How Developers See It. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, 1028–1038.

37. Oleksii Kononenko, Olga Baysal, Latifa Guerrouj, Yaxin Cao, and Michael W. Godfrey. 2015. Investigating Code Review Quality: Do People and Participation Matter?. In *Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME '15)*. IEEE Computer Society, 111–120.

38. Thomas D. LaToza, Gina Venolia, and Robert DeLine. 2006. Maintaining Mental Models: A Study of Developer Work Habits. In *Proceedings of the 28th International Conference on Software Engineering (ICSE '06)*. ACM, 492–501.

39. Mark S. Litwin. 1995. *How to Measure Survey Reliability and Validity*. SAGE Publications.

40. Mika V. Mantyla and Casper Lassenius. 2009. What Types of Defects Are Really Discovered in Code Reviews? *IEEE Trans. Softw. Eng.* 35, 3 (May 2009), 430–448.

41. Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. 2014. The Impact of Code Review Coverage and Code Review Participation on Software Quality: A Case Study of the Qt, VTK, and ITK Projects. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR '14)*. ACM, 192–201.

42. Kumiyo Nakakoji, Yasuhiro Yamamoto, Takahiro Suzuki, Shingo Takada, and Mark D. Gross. 1998. From Critiquing to Representational Talkback: Computer Support for Revealing Features in Design. *Knowledge-Based Systems* 11, 7 (Dec. 1998), 457–468.

43. Mehrdad Nurolahzade, Seyed Mehdi Nasehi, Shahedul Huq Khandkar, and Shreya Rawal. 2009. The Role of Patch Review in Software Evolution: An Analysis of the Mozilla Firefox. In *Proceedings of the Joint International and Annual ERCIM Workshops on Principles of Software Evolution and Software Evolution Workshops (IWPSE-Evol '09)*. ACM, 9–18.

44. Teade Punter, Marcus Ciolkowski, Bernd G. Freimut, and Isabel John. 2003. Conducting On-line Surveys in Software Engineering. In *Proceedings of the 2003 International Symposium on Empirical Software Engineering (ISESE '03)*. IEEE Computer Society, 80–88.

45. Caitlin Sadowski, Jeffrey van Gogh, Ciera Jaspan, Emma Söderberg, and Collin Winter. 2015. Tricorder: Building a Program Analysis Ecosystem. In *Proceedings of the 37th International Conference on Software Engineering (ICSE '15)*. IEEE Computer Society, 598–608.

46. N. Shyamsundar and Rajit Gadh. 2001. Internet-based Collaborative Product Design with Assembly Features and Virtual Design Spaces. *Computer-Aided Design* 33, 9 (Aug. 2001), 637–651.

47. Janet Siegmund, Norbert Siegmund, and Sven Apel. 2015. Views on Internal and External Validity in Empirical Software Engineering. In *Proceedings of the 37th International Conference on Software Engineering (ICSE '15)*. IEEE Computer Society, 9–19.

48. Patanamon Thongtanunam, Shane McIntosh, Ahmed E. Hassan, and Hajimu Iida. 2015. Investigating Code Review Practices in Defective Files: An Empirical Study of the Qt System. In *Proceedings of the 12th Working Conference on Mining Software Repositories (MSR '15)*. IEEE Computer Society, 168–179.

49. Pradeep K. Tyagi. 1989. The Effects of Appeals, Anonymity, and Feedback on Mail Survey Response Patterns from Salespeople. *Journal of the Academy of Marketing Science* 17, 3 (June 1989), 235–241.

50. Alonso H. Vera, Thomas Kvan, Robert L. West, and Simon Lai. 1998. Expertise, Collaboration and Bandwidth. In *Proceedings of the 16th International Conference on Human Factors in Computing Systems (CHI '98)*. ACM, 503–510.

51. Wikipedia. 2017. Gerrit (software). (2017). Accessed Feb 2017.

52. Yunwen Ye, Yasuhiro Yamamoto, and Kumiyo Nakakoji. 2007. A Socio-technical Framework for Supporting Programmers. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC-FSE '07)*. ACM, 351–360.