Empirical Software Engineering (2023) 28:5
https://doi.org/10.1007/s10664-022-10232-4

Can static analysis tools find more defects?

A qualitative study of design rule violations found by code review

Sahar Mehrpour[1] · **Thomas D. LaToza**[1]

Accepted: 28 August 2022

Abstract
Static analysis tools find defects in code, checking code against rules to reveal potential defects. Many studies have evaluated these tools by measuring their ability to detect known defects in code. But these studies measure the current state of tools rather than their future potential to find more defects. To investigate the prospects for tools to find more defects, we conducted a study where we formulated each issue raised by a code reviewer as a violation of a rule, which we then compared to what static analysis tools might potentially check. We first gathered a corpus of 1323 defects found through code review. Through a qualitative analysis process, for each defect we identified a violated rule and the type of Static Analysis Tool (SAT) which might check this rule. We found that SATs might, in principle, be used to detect as many as 76% of code review defects, considerably more than current tools have been demonstrated to successfully detect. Among a variety of types of SATs, Style Checkers and AST Pattern Checkers had the broadest coverage of defects, each with the potential to detect 25% of all code review defects. We found that static analysis tools might be able to detect more code review defects by better supporting the creation of project-specific rules. We also investigated the characteristics of code review defects not detectable by traditional static analysis techniques, which to detect might require tools which simulate human judgements about code.

Keywords Defect classification · Rule checker classification · Code review · Static analysis · Static analysis tools

Communicated by: Emerson Murphy-Hill

This work was supported in part by the National Science Foundation under grant NSF CCF-1703734.

✉ Sahar Mehrpour
smehrpou@gmu.edu

Thomas D. LaToza
tlatoza@gmu.edu

[1] Department of Computer Science, George Mason University, Fairfax, VA, USA

Published online: 08 November 2022

1 Introduction

Modern code review is central to identifying an important and diverse range of software defects (Bacchelli and Bird 2013; Wagner et al. 2005; Runeson et al. 2006). But it has long been envisioned that more of this work might be done by tools which can reliably, quickly, and accurately identify defects as soon as they occur or even offer automated support for fixing defects (Mao et al. 2016). In pursuit of this vision, Static Analysis Tools (SATs) have been devised using a wide range of static analysis techniques to detect behavioral defects, code smells (van Emden and Moonen 2002), code style violations, code clones, build issues, poor test coverage, and other types of defects. After being identified by tools, most defects are addressed by developers (Balachandran 2013), demonstrating the value of the defects found by these tools to developers (Habib and Pradel 2018). Beyond detecting a defect, some tools suggest a cause or propose fixes, as many developers expect (Johnson et al. 2013). For example, Style Checkers propose edits to fix style defects and Syntax and Semantic Analyzers propose corrections to address syntax errors.

To assess the state of this vision, SATs are often evaluated in their ability to detect known defects. For example, studies have systematically compared the lines with defects with the defect lines reported by FindBugs, JLint, and PMD, revealing that 35% to 95% of defects reported through issue trackers might be detected (Thung et al. 2012). Manually mapping known defects in Defects4J to the defects found by Error Prone (Aftandilian et al. 2012), Infer (Calcagno et al. 2015), and SpotBugs (the successor to FindBugs (Hovemeyer and Pugh 2004)) revealed that these tools could detect 4.5% of defects (Habib and Pradel 2018). A comparison of defects found in code review to those found by PMD revealed that it could detect 16% of issues. By adding project-specific rules, it might detect an additional 17% of the issues (Singh et al. 2017).

While these studies offer important evidence about the effectiveness of contemporary tools, they offer less insight into the potential to continue to improve the effectiveness of SATs. Of the 70% of code review defects (Singh et al. 2017) or 95% of known Defects4J defects *not* detectable by current tools (Habib and Pradel 2018), what types of static analysis techniques might be necessary to identify these defects? Answers to these questions offer important evidence helping motivate investments in creating tools, help to prioritize investment in specific techniques based on their potential impact, and identify specific use cases where these tools might be successfully applied.

Answering these questions requires investigating the *potential* of future SATs to detect defects. Rather than examine which defects can be found by existing tools, these questions requires examining the type of defects which may exist and the corresponding types of static analysis tools which might be required to detect these defects, whether these tools exist today or do not. In particular, answering these questions requires an alternative approach to assessing static analysis tools. Rather than run static analysis tools that exist today, it is necessary to instead examine the *defects* which exist today and then carefully consider how these might potentially be detected by current and future tools.

In this paper, we investigate the potential of SATs to find *more* defects. Rather than conservatively measure which defects can be detected by current tools, we are optimistic, characterizing an upper bound of defects which might be deterministically detectable through static analysis techniques. We focus on the defects which may be identified through code review, which identify a broad range of functional and non-functional issues (Beller

et al. 2014). By examining the reasoning code reviewers give to authors through their code review comments, we formulate each issue raised by a code reviewer as a violation of a *rule*. We then compare each rule against the rules which SATs might check, generalizing across similar specific tools to focus on the expressiveness of the underlying techniques behind these tools. In some cases, this may include rules that cannot, in fact, yet be checked by any current tool. For example, checking these rules might require a more precise static analysis than is currently available. In other cases, these defects may correspond to rules that *could* be checked by current tools, but which are simply not implemented. In this way, we examine the potential of static analysis techniques to find defects, rather than their current realization by contemporary tools or the rules developers choose to write.

We also examine the characteristics of defects which *cannot* be detected by static analysis tools, even assuming current techniques reach their full potential. These defects are characterized by issues which require human judgement to identify. For example, a code reviewer might find that, following inspection of the uses of a constant, its identifier is inaccurate. Or a developer might decide that, after examining a new event, it should be logged in a different way. These defects reflect issues which cannot be deterministically identified through the *formal channel* of code (Casalnuovo et al. 2020), encompasses information specifying computer execution and derives its meaning from the semantics of code. In these cases, human developers make judgements, using information in the *natural language channel* of code (Casalnuovo et al. 2020) such as identifiers, comments, or artifacts like documentation, as well as their own knowledge. While traditional static analysis tools are limited to considering only information in the formal channel, future ML or NLP based tools might potentially detect some of these defects through use of the natural language channel.

To investigate these questions, we examined the characteristics of defects found in code review. We first collected 1323 review comments from 493 pull requests across projects written in 33 different programming languages. We then used qualitative data analysis to systematically identify a static analysis technique which might detect the issue raised in each of the review comments. To organize the types of static analysis-based tools which might be relevant, we created a new taxonomy characterizing the representation of code used to check for defects (e.g., abstract syntax tree, program execution, string literals), the origin of the rule (e.g., programming language, project conventions), and the consequences of its violation (behavioral changes or code quality). We also classified each defect based on their impacts on the project (e.g., maintainability, user interface). Using these two taxonomies, we then coded each of the 1323 review comments.

We found that current static analysis techniques using the formal channel may be capable of detecting 76% of the defects found through code reviews. This fraction is considerably higher than the 16% detected by PMD (Singh et al. 2017) and 4.5% detected by Error Prone, Infer, and SpotBugs (Habib and Pradel 2018), offering evidence that current SATs might find more defects if used to their full potential. SATs leveraging simpler program representations, such as AST Pattern Checkers and Style Checkers, were the most broadly applicable, each identifying 25% of the defects reported, and together accounted for more than two thirds of the defects which might be found by SATs. Examining these defects, we identified specific features required to detect them. We also found that code review defects not detectable by traditional static analysis techniques are mainly maintainability and implementation defects.

2 Related Work

Our work builds on prior work proposing defect taxonomies, examining the ability of SATs to detect defects, examining developers' experience with SATs, using machine learning techniques to detect defects, and examining the practices of code review.

A defect may be defined narrowly as code which may lead to a system failure (e.g., Laitenberger 1998; Wiegers 2002; Burnstein 2002; Group and et al 2010; Runeson et al. 2006) or more broadly as code that reduces the quality of the codebase (Gilb et al. 1993; Humphrey 1995; Laitenberger and DeBaud 2000; Runeson and Wohlin 1998). Mäntylä and Lassenius (2009) broadly define defects as deviations from the viewpoint of a code reviewer. We adopt this definition in this paper. Defect taxonomies generally focus on describing the causes of defects (Basili and Selby 1987; Runeson et al. 2006). The Orthogonal Defect Classification (ODC) (Chillarege et al. 1992) classifies defects along two dimensions: defect type (e.g., function, interface, checks) and defect trigger, specifying the conditions under which the defect surfaces based on the development phase (e.g., review, test, deployment). One study of defects found in code reviews identified three categories: evolvability defects which make code hard to read and maintain, functional defects which result in incorrect behavior, and false positives (Mäntylä and Lassenius 2009), influencing a later taxonomy (Beller et al. 2014). Similarly, the General Defect Classification (GDC) categorizes warnings generated by Static Analysis Tools into functional and maintainability defects.

Static Analysis Tools (SATs) encompass a wide range of tools. SATs have been used to detect violations of coding style (e.g., CheckStyle (2004); JSlint (Crockford 2011), JSNose (Fard and Mesbah 2013)), detect instances of 'bug patterns' (e.g., PMD (Copeland 2005) and FindBugs (Hovemeyer and Pugh 2004)), detect violations of architectural style rules (e.g., Structure101 (2019), SAVE (Knodel and Popescu 2007), ArchJava (Aldrich et al. 2002), Darcy (Ghorbani et al. 2019)) or design rules (e.g., ActiveDocumentation (Mehrpour et al. 2019)), identify code smells (Sharma and Spinellis 2018), or detect behavioral defects (e.g., CPAchecker (Beyer and Keremoglu 2011)). Thung et al. (2012) investigated 3 defect detection tools, FindBugs, JLint, and PMD, on 3 programs, Lucene, Rhino, and AspectJ. For 200 fixed defects, they compared the lines containing a fix with warnings generated by the tools and found that between 1.9% to 50% of lines relating to defects were missed by tools, with numbers varying greatly between projects and how completely the defect was detected. Habib and Pradel (2018) investigated the ability of Error Prone (Google), Infer (Facebook), and SpotBugs (the successor to FindBugs) to detect 594 real-world defects across 15 software projects. They found that the tools were able to find 4.5% of these defects. The undetected defects were mainly related to project-specific rules.

A number of studies have examined how developers use SATs. One study found that 59% of open-source projects use SATs, with projects written in dynamically-typed languages benefiting most (Beller et al. 2016). An early study at Google found that while the warnings generated by FindBugs were not causing serious problems in production, early identification makes them low-cost to fix (Ayewah and Pugh 2010). When SATs break the build, in most cases developers fix the code rather than changing the configuration of the tool (Zampetti et al. 2017). Developers prioritize warnings of SATs based on development contexts (Vassallo et al. 2018). A number of studies have found usability challenges with SATs. Developers face barriers using SATs due to poor presentation of output, lack of support for collaborative environments to share tool settings, complexity using customization, unclear results, and disjointed process (Johnson et al. 2013). Tools often have a high false positive rate, and developers wish to have the ability to prioritize errors, suppress warnings, and get accurate results (Christakis and Bird 2016). Compiler error messages may

be improved by being written using new and more usable structures that more effectively meet developers' information needs (Barik et al. 2018). Disabling errors and warnings not marked as useful can help maintain developer trust in tools (Sadowski et al. 2018b).

More recently, research has begun to explore the use of machine learning (ML) and natural language processing (NLP) techniques to detect defects in code (Shafiq et al. 2021). ML-based defect detection tools first build a model using traditional classifiers or neural networks, training the model with data from code from different projects or older versions of the same project. Next, the tools compare the target source code against the computed model to find defects or predict the existence of defects. ML-based tools may build models from semantic information in code such as comments (Huo et al. 2018), AST tokens such as identifier names and operators (Wang et al. 2020), or source code changes (Wang et al. 2020). This can then be used to identify defects, such as missing authentication for smtp connections (Huo et al. 2018) or missing IOException handling when reading files (Wang et al. 2020). Other ML-based tools use models built from code metrics such as the lines of code or number of methods (Nam and Kim 2015) to predict the existence of defects in code. Other ML-based tools transform code into intermediate representations before building models. For example, DeepBugs (Pradel and Sen 2018) maps identifiers to semantic representation vectors using NLP techniques and neural networks and detects name-based defects such as incorrect orders of input parameters. Some tools analyze semantic information in code such as argument names using NLP techniques such as string distance and detect defects like incorrect order of input parameters (Rice et al. 2017).

Code review remains one of the most important techniques for finding defects, despite being less effective and efficient compared to techniques such as testing (Runeson et al. 2006). Modern Code Review is a regular, informal, tool-based process (Bacchelli and Bird 2013). Developers review code to improve understandability and maintainability (Sadowski et al. 2018b), find defects, transfer knowledge, offer alternative solutions, and track rationale (Bacchelli and Bird 2013; Ebert et al. 2018). Developers sometimes, but generally rarely, discuss design during code review (Viviani et al. 2018). Code review practices have commonality, even across projects and organizations (Rigby and Bird 2013). Code reviewers need information including rationale and code context (Pascarella et al. 2018).

Researchers have used code reviews as a lens with which to understand defects and fixes. One study found that 75% of found defects are evolvability defects, most of which cannot be fixed directly by tools (Mäntylä and Lassenius 2009). Another study again found that 75% of the changes made after code review were related to the evolvability of code, and 35% of review comments are discarded (Beller et al. 2014). Other studies used code reviews to investigate the role of defect detectors in software development. One study found that defects found by SATs are different from those found by tests, but are a subset of those found in code review (Wagner et al. 2005). Another study found many code review defects concern code improvements and suggested the need for tools to eliminate these defects and free time for finding more important defects (Bacchelli and Bird 2013). One study found that during code reviews, some of the warnings generated by well-known SATs are resolved, and that using SATs helps in fixing these warnings and thus speeds up the code review process (Panichella et al. 2015). Another study found that PMD could be used to identify 16% of the defects found through code review and that, by implementing and integrating 4 new rulesets, can cover an additional 17% of defects (Singh et al. 2017). Developers choose to approve 93% of the review comments generated by PMD, FindBugs, and Checkstyle (Balachandran 2013). Contributors repeatedly introduce the same types of manually detectable issues, while they repeat automatically detectable issues at most 3 times (Ueda et al. 2018).

3 Analyzing Review Comments

In this paper, we consider the potential for future Static Analysis Tools (SATs) that use the formal channel (Casalnuovo et al. 2020) to detect defects. To explore the ability of SATs to detect *more* defects, we sought to investigate:

RQ1: What types of existing SATs are necessary to identify defects?
RQ2: What features in SATs are necessary to detect defects?
RQ3: Which defects cannot be detected by only using the formal channel?

To answer these research questions, we conducted a qualitative study of defects identified in review comments. We manually examined defects identified, investigating the potential to prevent each defect from occurring by creating a rule which could be checked by a SAT. Throughout this process, we considered in detail a broad range of tools which use the formal channel and might be relevant and the features necessary in these tools to detect each defect. This enables examining the potential impact of increasing the power of existing analysis techniques, as well as highlighting defects which would require new forms of analysis beyond the traditional use of the formal channel. Running existing SATs would not answer these questions, as it would report issues discovered by a tool as it is configured and exists today.

In this paper, we adapt a broad definition of a defect as a deviation of code from a quality standard from the point of view of a reviewer (Mäntylä and Lassenius 2009). This encompasses a wide variety of defects, including defects that alter the behavior, quality, and performance of code. We exclude documentation defects, which have been studied elsewhere (Aghajani et al. 2019)).

We analyzed defects found in code reviews and examined which SATs might detect these defects and how these defects impact their projects. To organize the types of SAT which might be relevant, we require a taxonomy of tools. We first looked at existing defect classifications. Defect classifications classify defects based on their technical properties (function, interface, checks, assignment, timing, building and merging, documentation, and algorithms) (e.g. Board 1993; Chillarege et al. 1992; Mäntylä and Lassenius 2009; Beller et al. 2016), their severity and impact on the user (e.g. Thelin et al. 2003), and the artifacts from which the defects originate (e.g. Runeson et al. 2006). As these taxonomies do not categorize the analysis tools themselves, we created a new taxonomy of SATs for defect detection, characterizing the representation of code used to check for defects, the origin of the rule, and the consequences of violations. In addition, we created a second taxonomy to characterize the type of impact that each violation had, which adapted several categories from existing taxonomies.

3.1 Data Collection

The process of data collection and cleaning is illustrated in Fig. 1. To obtain an initial corpus of code review comments, we first examined code reviews posted in pull requests of public repositories. To encompass a broad range of projects with varying programming languages and levels of complexity, we selected the GHTorrent (Gousios 2013) dataset, a widely-used dataset of GitHub data (e.g., Brunet et al. 2014). We examined a subset of the GHTorrent dataset containing 36,185 pull requests (PRs) from public GitHub repositories on May 5, 2019. As the original dataset includes too many PRs to manually inspect, we systematically selected a subset of PRs for further analysis. We first used GitHub labels, provided by contributors, to identify PRs that might potentially describe defects. Each GitHub pull request

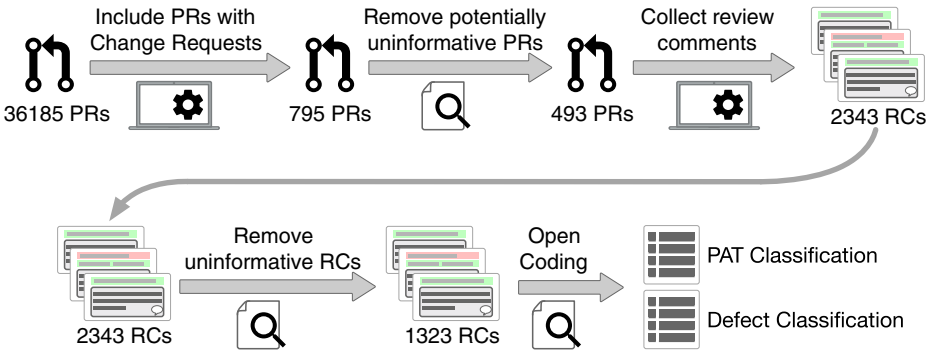

Fig. 1 We employed a systematic process to identify defects from informative review comments (RC) in pull requests (PR) and code each defect

(PR) may include one or more *reviews*, each labeled with *approve* (change approved by the reviewer), *comment* (feedback by the reviewer), or *change request* (feedback to address before merging). For our study, we only considered change requests, as they were most likely to describe defects to be addressed. 795 PRs with at least one change request were identified and considered for further analysis. There are several potential explanations for the infrequency of PRs containing a review with the change request label. First, many PRs lack reviews and contain only comments. Second, change requests contain feedback needed to be addressed for merging the PR and so are made before merging the PR. Considering the high number of rejected PRs (20% to 93% Silva et al. 2016), many PRs are not considered by reviewers and do not receive change request feedback.

We next performed two rounds of data cleaning. In the first round, we manually inspected each PR and filtered out PRs which were irrelevant or missing required information. We removed PRs with non-English comments with reviews from the github-learning-lab (used for learning purposes) which related to homework assignments, which focused on wording and grammatical issues in documents, and with inaccessible repository links. All links existed at the time of data collection, but some later became unavailable. After this process, 493 PRs remained. These pull requests spanned approximately 400 distinct projects in at least 33 different programming languages.

In the second round of data cleaning, we identified and filtered review comments. We first systematically collected the discussions of change requests for each of the 493 PRs through the GitHub API. In total, 2343 review comments were collected. We manually analyzed each review comment and excluded from our analysis review comments that were not detailed enough for analysis (Section 4). This included review comments with no text, which included directions for GitHub actions (e.g., merging), which were not written in English, included only non-informative text such as emojis, were generated by automated tools (e.g., bots), were not directly associated with specific lines of code, or which lacked context, code, or sufficient explanation. After this process, 1323 review comments remained. Our dataset is publicly available.[1]

[1] https://doi.org/10.6084/m9.figshare.14925222

3.2 Analysis

To identify the relationship between SATs and the defects they might find, we identified a rule for each defect. Rules express a constraint on a specific representation of code and its execution, including its Abstract Syntax Tree (e.g., Copeland 2005, clone detectors), execution paths (e.g., code coverage tools for test suites JCov 2014), and string literal values (e.g., tools using regular expressions to check for valid filepaths). Rules may express universal constraints (e.g., do not use a variable after it has been freed, avoid `static` fields) or project-specific constraints (e.g., use loggers instead of `print`). Rule constraints may describe prohibited or required conditions in code (Kruchten 2004), which may enable tools to use rule violations to identify defects as well as suggest fixes (Section 8).

We first mapped each defect described in code review comments to a rule which might prevent the defect. To formulate rules, we used all available artifacts and information in the review comments. In addition to the text of the comment, we considered other artifacts such as previous or followup discussions (if they existed), the code to which the comment applied, and the final code commit (if accessible) to see how the feedback was addressed. For example, the review comment in Fig. 2 describes a defect involving a misplaced method which should be relocated to another file. From the name of the proposed file, it can be inferred that the file contains utility methods. Another available artifact is the implementation of the method, revealing that the method only uses the input arguments to read data (using another utility method) and writes the sorted data using another input argument. Using this information, we can describe this defect as a violation of a rule: "If a method uses only its arguments and other utility methods, then it should be located in a utility class." While we cannot quantify how frequently rules apply, we found many rules which were

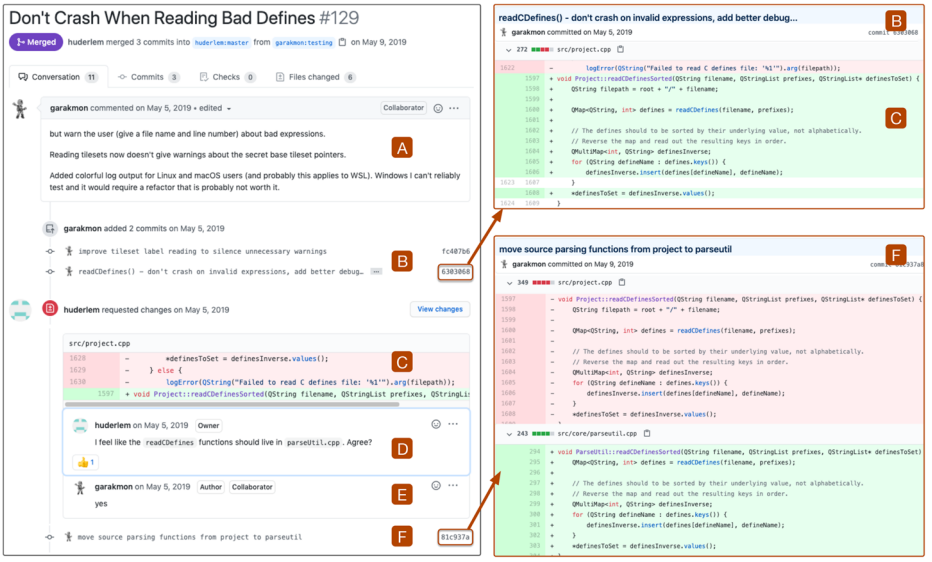

Fig. 2 We mapped each review comment to a rule. To do so, we examined artifacts and information including the (A) the description of the PR, (B) committed changes in the PR before the review, (C) faulty lines of code, (D) the review comment, (E) followup discussions, and (F) changes made after the review

violated several times in the same PR. Review comments also referenced other review comments to describe how to fix an issue. This suggests that some rules are not one-off and reflect recurring issues.

Next, we mapped the formulated rules to SATs. To find this mapping, we employed a systematic qualitative data analysis process (Saldaña 2015; Seaman 1999), adapting methods previously used to study code reviews (Sadowski et al. 2018b). We used a two-step coding process (Saldaña 2015) to identify categories of SATs and to map each review comment to a SAT.

In the first step, we applied open coding to generate a list of tool definitions. We began with an initial list of well-known SATs that check rules against code such as clone detectors, dead code detectors, and linters and created definitions for these categories. As we applied these codes, we created codes for new tool types when these were not applicable. The tool categories include both the means and the purpose of the analysis. We considered three orthogonal dimensions of tools as well as the purpose of tools to categorize SATs (defined in Section 4 and Table 1). We considered SATs independent of programming language (e.g., language-specific style checkers such as JSLint (Crockford 2011) as Style Checkers). We ignored tools that do not check rules, such as those that require developers to write code (e.g, keyword programming tools Little and Miller 2007).

In the second step, we applied focused coding, grouping similar tools identified in the previous step into categories through an iterative process. We identified 12 types of SATs (Fig. 4). Next we annotated review comments by extracting rules' properties and matching them against the dimensions and purposes of SATs. For example, the rule we identified for the issue in Fig. 2 requires that utility methods, which only use data from their arguments and only call other utility methods, be located in utility classes. To check this rule, method bodies may be examined for the variables referenced (i.e., only arguments and not global variables or fields) and the methods invoked (only other utility methods). These checks might be implemented through a tool which examines the AST of this code. Therefore, we labeled this issue as one which might be detected by an AST Pattern Checker (Section 4). Next, we used the final codes to annotate each review comment with tools able to check them (closed coding).

To investigate the impact of the defects on their projects, we built a second taxonomy. We followed a similar approach and first applied open coding on the rules for each defect identified from review comments. We used open coding and annotated the data in two phases. In the first phase, the first author applied descriptive coding and summarized each review comment in a word or a short phrase. In the second phase, all authors discussed the theme of the labels obtained and applied pattern coding iteratively. In each phase, authors discussed relationships between labels to clarify differences and grouped similar themes. To ensure consistency, we established coding guidelines. For example, if a reviewer explicitly asked for a "fix," we coded the defect as an implementation defect. Or if the reviewer asked for "consistency," we coded the defect as code quality. When a defect may belong to several defect types, we considered the most specific as the type of the defect. For example, we always considered defects involving the user interface as user interface defects, defects involving the performance of the codebase as performance defects, and defects involving test suites as test suite defects. We identified 7 defect types, identifying distinct impacts of a defect on a project (Fig. 3).

To ensure the reliability and repeatability of our classifications, we conducted an inter-rater reliability check. Two authors individually labeled 3 sets of 70 randomly selected review comments using code definitions. After each round, the authors compared the labels, identified divergences, and clarified the definitions to refine the scope of labels. Finally, the

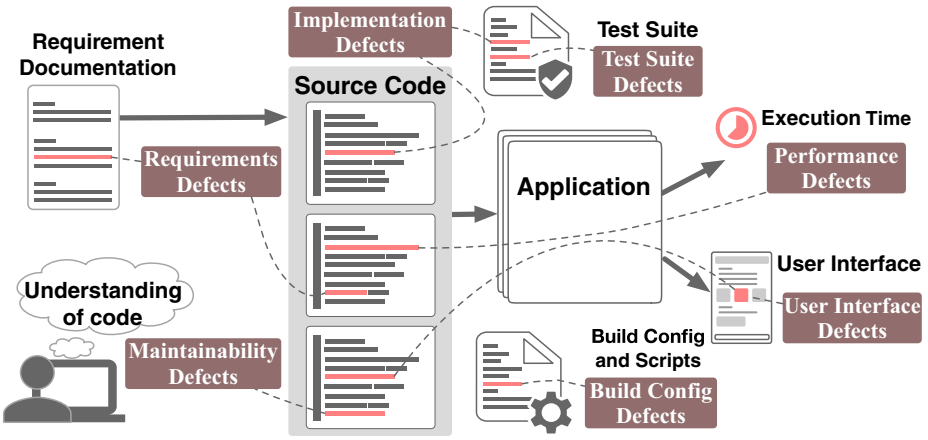

Fig. 3 We classified defects based on their impacts on the project

authors applied the coding scheme to an additional 70 randomly selected review comments. The calculated Cohen's κ for the SAT labels was $\kappa = 0.53$, showing that the agreement between the two raters is "Moderate" (Landis and Koch 1977). For the defect classification labels, the final calculated Cohen's κ was 0.68, showing that the agreement between the two raters is "Substantial" (Landis and Koch 1977). Disagreements mainly arose from differing interpretations of the reviewers' intent when the artifacts available in the review comments were limited.

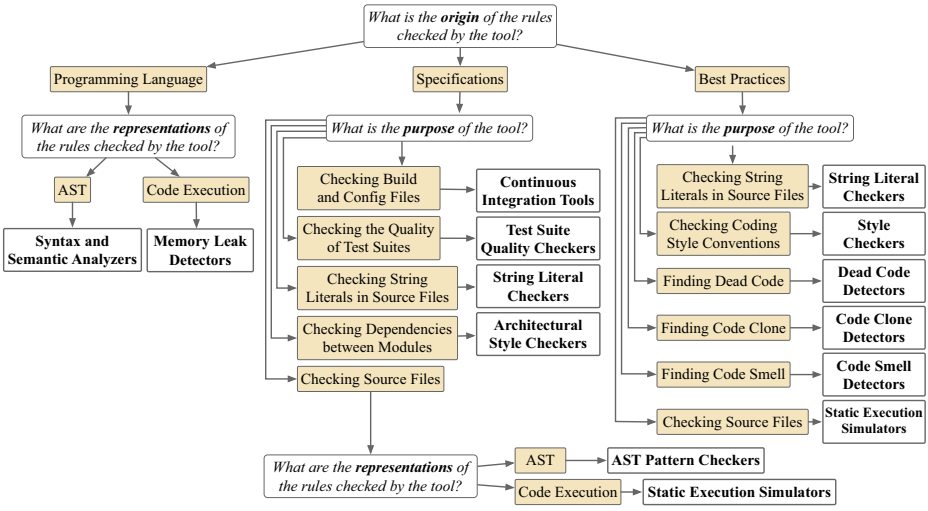

Fig. 4 Our SAT taxonomy categorizes the types of SATs based on the **origin** of the rules they check, the **purpose** of the tool, and the **representations** of the rules they check (nodes with italic text)

4 SAT Classification

We categorize SATs in our taxonomy through three orthogonal dimensions and their purpose (Fig. 4). The three orthogonal dimensions are Representation, Origin, and Consequence. **Representation** specifies whether rules are represented by checking an *AST* representation of the source code, checking *String* literals in source files, or *Code Execution* and tracking and monitoring paths in code. **Origin** indicates whether rules originated in the definition of the programming *Language*, are imposed by project *Specifications*, or are generated by *Best Practices*. **Consequence** indicates the impact of violations on code, by decreasing the *quality and maintainability* or by impacting its *behavior* by changing its output. We classified SATs first based on the Origin dimension, and next based on the purpose of the tools (e.g., finding dead code or finding code clones). In the last step, SATs are classified based on the Representation of the rules checked by the tools (Fig. 4). We also examined the Consequences of rules checked by SATs, but this dimension did not differentiate between any additional types of tools. Many real world static analysis tools, such as SonarQube (SonarSource 2022), Coverity (Bessey et al. 2010), Tricorder (Sadowski et al. 2015), and Infer (Calcagno et al. 2015), combine several of these categories together.

Here we define each SAT category (Table 1). As many SATs encompass hundreds of related tools (e.g., Static Execution Simulators), we illustrate each with seminal examples of tools.

Style Checkers ensure the readability of code by checking code style conventions (e.g., CheckStyle 2004). Violations decrease code quality. Rules may be expressed through constraints on a program's AST, through formatting and style conventions for comments, or for typos in comments and identifiers.

Table 1 Our SAT taxonomy includes 12 categories of tools. Each is specified through three dimensions: Representation, Origin, and Consequences. (A: AST, CE: Code Execution, ST: Strings, L: Language, SP: Specifications, BP: Best Practices, CQ: Code Quality, B: Behavioral). The ∼ symbol indicates indirect influence

	Representation			Origin			Consequence	
SAT Categories	A	CE	ST	L	SP	BP	CQ	B
Style Checkers	✓		✓			✓	✓	
Continuous Integration Tools	✓	✓	✓		✓			✓
Static Execution Simulators		✓		✓	✓		✓	✓
Architectural Style Checkers	✓	✓			✓		✓	
Test Suite Quality Checkers	✓	✓	✓		✓		∼	∼
Dead Code Detectors	✓	✓				✓	✓	
Code Clone Detectors	✓	✓				✓	✓	
Syntax and Semantic Analyzers	✓			✓				✓
String Literal Checkers			✓		✓	✓	✓	✓
Code Smell Detectors	✓	✓				✓	✓	
Memory Leak Detectors		✓		✓				✓
AST Pattern Checkers	✓				✓		✓	✓

Continuous Integration Tools ensure a successful build and integration process of components, such as by checking rules related to the availability and compatibility of libraries and the correctness of configuration settings in build and configuration files (e.g., Jenkins 2019, Travis CI 2019). These rules may be expressed as constraints on an AST representation, the execution of code, or on string literals. Rules are project-specific, and violations change the program's behavior.

Static Execution Simulators check rules by data flow analysis, control flow analysis, model checking, type checking, abstract interpretation, separation logic, or symbolic execution (e.g., CPAchecker Beyer and Keremoglu 2011). They may generate representations of execution such as call graphs or dependency graphs (e.g., Reacher LaToza and Myers 2011). Rules are expressed through project-specific specifications or programming languages, and checked through analysis of the code execution. Violations impact the behavior or quality of code.

Architectural Style Checkers check the as built software architecture of code against the intended as designed architecture, and may include rules that encompass many components (e.g., ArchJava Aldrich et al. 2002). They impose constraints on the names and locations of elements as well as their dependencies and communication. Rules are specified per-project and may be checked against the AST or by code execution.

Test Suite Quality Checkers check the quality of test suites, such as by evaluating code coverage or testing critical inputs (e.g., JCov 2014). Rules are expressed through constraints on the AST of tests, the execution behavior of tests, or the allowable values of string literals. Rules are project-specific, and violations may indirectly impact the program's behavior and code quality.

Dead Code Detectors detect code fragments that are never executed because they are unused or unreachable (e.g., JSNose Fard and Mesbah 2013, DUM Romano et al. 2016).

Code Clone Detectors identify contiguous segments of source code that are syntactically and semantically similar (Svajlenko and Roy 2015) (e.g., CCFinder Kamiya et al. 2002). Code clones include (1) identical code fragments, (2) identical code fragments with different user-defined identifiers and literal values, (3) similar code fragments with statements added, removed, or changed, and (4) semantically similar code fragments implementing the same functionality (Roy and Cordy 2007). The presence of code clones is often considered poor coding practice, violating code quality standards. Code Clone Detectors detect code clones by analyzing ASTs or code execution.

Syntax and Semantic Analyzers check code for conformance to a programming language's syntax and semantics. This includes constructing the AST of the program and checking it against the language grammar (syntax) as well as type checking and verifying the consistency of the AST with rules defined by the programming language (semantic analysis). Violations of these rules impact the behavior of the code.

String Literal Checkers check for the compliance of string literals in code with standard formats, such as for log messages or addresses of external hosts (e.g., Regex checkers). Rules are project-specific, and may be specified through grammars or regular expressions. Violations may impact the quality or behavior of code.

Code Smell Detectors identify defects that violate best practices and decrease the maintainability of code (van Emden and Moonen 2002) (e.g., PMD Copeland 2005, Findbugs Hovemeyer and Pugh 2004, JSNose Fard and Mesbah 2013). They may rely on rules concerning the program's AST or execution. In some taxonomies, code smells also include code clones and dead code fragments. In our taxonomy, we separate these from code smells.

Memory Leak Detectors identify memory that is allocated but not released (e.g., SABER Sui et al. 2012, LeakChecker Yan et al. 2014). For example, a rule may specify where a specific variable must be allocated and then later released. Rules may be checked through code execution, are imposed by programming language semantics, and their violation impacts program behavior.

AST Pattern Checkers check code against rules expressible through a pattern which may be represented through the AST of a program. These rules are specified by project specifications and affect the behavior or the quality of the code. AST Pattern Checkers may check rules which require the presence of code, prohibit code, or express alternatives. If a rule is checkable by both AST Pattern Checkers and another category of tools (e.g., Architectural Style Checkers), we label the rule with the latter category.

4.1 Defects Not Detectable by SATs

Checking some rules required additional information beyond the information found in the formal channel used by SATs. These rules required human judgement, which might be based on information from the natural language channel in code found in identifiers or comments, in design documents, or from a developers' own knowledge and judgement. While static analysis tools using the formal channel alone are unable to check these rules, future tools using the natural language channel might potentially be able to check some of these rules.

5 Defect Classification

We identified 7 types of defects; *Requirements*, *User Interface*, *Implementation*, *Maintainability*, *Performance*, *Test Suite*, and *Build Config* (Fig. 3). The Maintainability and Implementation categories are similar to the Evolvability and Functional defects identified by Mäntylä and Lassenius (2009).

Maintainability defects impact the quality, rather than the behavior, of code. Maintainability defects encompass defects universally applicable to all projects, such as code smells, as well as project-specific defects. For example, refactoring improves the readability of the code, and defects that require refactoring are Maintainability defects. This category includes a subset of Evolvability defects (Mäntylä and Lassenius 2009) that contains a broader range of defects that "make the code less compliant with standards, more error-prone, or more difficult to modify, extend, or understand".

Implementation defects (also known as Functional Defects Mäntylä and Lassenius 2009) occur when code does not satisfy its requirements.

Rules preventing implementation defects are defined per-project and their violations change the behavior of code. This category of defects contains a subset of Functional

defects (Mäntylä and Lassenius 2009) which "may cause system failure when the code is executed.".

Build Config defects encompass anything that may cause the build and integration of the codebase to break, such as missing scripts or inconsistent external libraries. This includes defects in scripts and config settings that configure how a compiler and other tools integrate source code into a running application. For example, incorrect local or external addresses of libraries are Build Config defects. Defects such as using methods from incorrect libraries does not break the build and thus they does not fall in this category.

Test Suite defects encompass all defects in test suites. These include incorrect and ineffective test suites, such as an incorrect assertion that makes the test suite incorrect or insufficient code coverage. As tests checks the correctness of applications, Test Suite defects may be related to behavioral defects. Tests can check code, and therefore, indirectly impact code quality.

User Interface defects encompass visual appearance and the ways in which elements behave. Defects impact the interface displayed to the user and the resulting usability of the application for the user. We label a defect as a *User Interface* defect if it directly changes the user interface and the way the user interacts with it. For example, an inappropriate element color is a User Interface defect. However, an incorrect implementation used to set a color (e.g., specificity, inheritance, or cascading in CSS) is not a User Interface defect.

Requirements defects involve missing consideration or misinterpretation of project requirements. If through review comments or code changes the contributor indicates that they understood the requirements but implemented them incorrectly, the identified defect is not a Requirements defect. We differentiate misinterpreting requirements from incorrectly implementing requirements. For example, if persisting specific data is a requirement, then keeping the data in a cache or disposing the data are misinterpretation and ignoring the requirements. An incomplete or incorrect implementation is an incorrect implementation of the requirements.

Performance defects encompass issues impacting runtime or memory usage, such as unnecessary computation or poorly optimized memory allocation.

For example, performance defects include unnecessary queries to the database which reduce application performance.

5.1 Comparison with Other Defect Taxonomies

Our taxonomy adapts some categories from existing defect taxonomies but differs in its goal. Our taxonomy focuses on the impact of the defect, while most existing defect taxonomies focus on the *cause* of the defect (Section 2). Our taxonomy has important similarities to ODC (Chillarege et al. 1992) and the classification of defects found in code reviews by Mäntylä and Lassenius (2009). ODC (Chillarege et al. 1992) has 8 main defect types based on the immediate cause of the defect (e.g., incorrect data validation, assignment errors). Our taxonomy instead organizes defects based on the impact of the defect on the project. For example, ODC does not explicitly identify *Maintainability* or *Performance* defects. The main difference in Mäntylä and Lassenius (2009) is the major categories, which are Functional and non-Functional defects.

Table 2 The percentage of defects which might potentially be detected (FC), by tool category and by defect type. The *Count* column is the number of defects. The *Overall %* column for All Defects lists the percentage of all defects found in review comments of that type. The *Tool Type %* column lists the percentage of defects found by the tool of that defect type. The *Overall %* of Defects Detectable by SATs column lists the percentage of detectable defects found in review comments

SAT Type and Defect Type	**All Defects**		**FC Defects**	
	Count	**Overall %**	**Tool Type %**	**Overall %**
All SATs	1009	76.27 %		100 %
AST Pattern Checkers	340	25.70 %		33.70 %
Maintainability	83		24.41 %	
Implementation	188		55.29 %	
Test Suite	8		2.35 %	
User Interface	33		9.71 %	
Requirement	20		5.88 %	
Performance	8		2.35 %	
Style Checkers	339	25.62 %		33.60 %
Maintainability	339		100 %	
Continuous Integration Tools	67	5.06 %		6.64 %
Build Config	67		100 %	
Static Execution Simulators	65	4.91 %		6.44 %
Maintainability	16		24.62 %	
Implementation	43		66.15 %	
Requirement	1		1.54 %	
Performance	5		7.69 %	
Architectural Style Checkers	60	4.54 %		5.95 %
Maintainability	57		95.00 %	
Test Suite	3		5.00 %	
Test Suite Checkers	29	2.19 %		2.87 %
Test Suite	29		100 %	
Code Clone Detectors	28	2.12 %		2.78 %
Maintainability	27		96.43 %	
Test Suite	1		3.57 %	
Dead Code Detectors	27	2.04 %		2.68 %
Maintainability	27		100 %	
Syntax and Semantic Analyzers	20	1.51 %		1.98 %
Implementation	20		100 %	
String Literal Checkers	20	1.51 %		1.98 %
Maintainability	10		50.00 %	
Implementation	10		50.00 %	
Code Smell Detectors	10	0.76 %		0.99 %
Maintainability	10		100 %	
Memory Leak Detectors	4	0.30 %		0.40 %
Implementation	4		100 %	

6 RQ1: What Types of Existing SATs are Necessary to Identify Defects?

To investigate the potential of SATs to find more defects, we first examined the defects which are detectable through the use of the formal channel (Casalnuovo et al. 2020) or *FC*. The formal channel of code encompasses information specifying computer execution and derives its meaning from the semantics of code. We found 12 types of SATs necessary to identify more defects (Sections 6.1 to 6.12). We found that AST Pattern Checkers and Style Checkers were the most broadly applicable, encompassing 25.70% and 25.62% of all defects, respectively. Our results are listed in Table 2.

6.1 AST Pattern Checkers

Among all tool types, AST Pattern Checkers had the broadest potential applicability (25.70% of all defects, 33.70% of FC defects) to detect defects. Many of the defects potentially detectable concern incorrect implementations of requirements or the maintainability of code.

Among all defects potentially detectable by AST Pattern Checkers, many involved violations of rules defined by implementation rules (55.29% of defects potentially detectable by AST Pattern Checkers). Some were caused by incorrect usage of existing code (e.g., incorrect method calls) or extra implementation (e.g., extra check for an input). For example, one defect required a specific method invocation for every instance of an object:

> Example 1. "we will want to emit the `clicked` signal no matter whether the topic is sensitive or not."
>
> If any instance of an object is created, then a specific method should be called regardless of the object properties.

Another frequent type of defect which might be detectable by AST Pattern Checkers involve maintainability defects (24.41% of defects potentially detectable by AST Pattern Checkers). These include code conventions and best practices (e.g., using '`equals()`' instead of '`==`' for comparing String literals) true across projects as well as project-specific rules. For example, one rule stipulated use of a logger instead of print to signal errors (Ex. 2).

> Example 2. "Remove `e.printStackTrace();` use logger instead."
>
> If an error is emitted, then it should be tracked using logger.

Other defects involved the user interface of the program (9.71% of defects potentially detectable by AST Pattern Checkers). These concerned violations of the intended visual design of the user interface or potential negative impacts for the usability of the software for end-users. For example, one review comment concerned a defect where an external link on the system website had an incorrect setting in which the browser was directed to leave the website. This setting impacted the usability of the system for the user and could be addressed by changing an HTML tag attribute (`target`). The review comment specified that this attribute should be applied in a specific set of enumerated circumstances (all external links).

Example 3. "please add a `target="_blank"` and a descriptive `title` tag to all the external links"
If an HTML tag contains an external link, then the tag should also include a specific property and a proper title.

Some potentially detectable defects negatively impacted performance or memory use (2.35% of defects detectable by AST Pattern Checkers). For instance, one defect involved persisted data that was retained in-memory. The reviewer specified that persisted data should only be retrieved on demand from the database rather than retained in memory (Ex. 4).

Example 4. "Why are we keeping these [persisted data] in memory at all? We could just as well load them on demand when a new `channelId` gets started"
If persisted data is used, it should be loaded on demand rather than retained in-memory.

6.2 Style Checkers

We found that Style Checkers had a high potential to detect defects (25.62% of all defects and 33.60% of all FC defects). Style Checkers check a wide range of maintainability rules, such as best practices and universal coding conventions. For example, reviews frequently surfaced defects involving violations of indentation and spacing conventions or universal naming conventions (Ex. 5).

Example 5. "Do not use underscore in method naming. Read [document on] java code convention. Method name should be `getTaskStatus` [instead of `get_task_status`]"
If an identifier is for a method, then it should not include underscores.

6.3 Continuous Integration Tools

Continuous Integration Tools might be used to detect defects that break the build and the integration of the codebase (5.06% of all defects). For example, these tools can detect whether incorrect versions of a library is used by checking configuration files (Ex. 6).

Example 6. "Please use the same version for spring `5.1.6.RELEASE` or `5.1.5.RELEASE`."
If a library is used, then a specific version should be used.

6.4 Static Execution Simulators

Static Execution Simulators might potentially be used to detect 4.91% of all defects. We found that many of these defects concern incorrect implementation of requirements or maintainability defects. These tools may also sometimes be relevant to detecting defects involving system performance. Static Execution Simulators can detect incorrect, incomplete, or extra implementations of requirements through code abstraction (66.15% of defects detectable by Static Execution Simulators). For examples, these tools can check if a variable

correctly holds one of a set of permitted values or if a temporal property holds for events in code. In Example 7, a defect in a system with publish/subscribe architecture can be detected by checking the order in which method calls occur.

> Example 7. "This doesn't really work, event is thrown before anything registers for it."
> If an event is emitted, there should be at least one subscriber.

Static Execution Simulators might also be used to detect maintainability defects (24.62% of defects potentially detectable by Static Execution Simulators). For example, a rule might describe how specifiers on mutability might be applied based on a variable's usage (Ex. 8).

> Example 8. "[It] looks like a `const` method to me."
> If a method does not mutate data in any fields, then it should be specified as `const`.

Static Execution Simulators might also be applied to performance defects (7.69% of defects detectable by Static Execution Simulators). Defects that impact runtime or memory usage might be specified by tracing values assigned to variables and fields. For example, static execution simulation might be used to determine whether a variable is needed by identifying where it is assigned values and how it is used (Ex. 9).

> Example 9. "... table index `i` and exact bit number `j` is known, why not to set it right away instead of packing it up in big number and unpack it again? If I check correctly this is only place where this macro is used."
> If some data is small and rarely used, then it should not be stored in large fields.

6.5 Architectural Style Checkers

Architectural Style Checkers might potentially be used to detect 4.54% of the defects in our dataset. These tools might check maintainability rules specifying allowed architectural styles. For example, a tool might check if classes within a component follows its policy on the correct level of accessibility (Ex. 10).

> Example 10. "Make the class modifier and all its queries as package local ... otherwise, you won't be able to access these."
> If a class belongs to a specific package, then all its elements should be package local.

They might also check rules concerning where elements should be located, specifying necessary refactorings (Ex. 11).

> Example 11. "Move this method to `CanvasDataService`, the service defines a set of related logic together with some enum fields, I see no reasons to keep them apart."
> If elements share similar logic, then they should be kept together in the codebase.

6.6 Test Suite Quality Checkers

Test Suite Quality Checkers (2.19%) might be used to find defects affecting the correctness and effectiveness of test suites, such as low code coverage or missing or redundant tests (Ex. 12).

> Example 12. "... please add one or more functional tests to cover this new flag."
>
> If new code is added, then there should be tests that cover it.

6.7 Code Clone Detectors

Code Clone Detectors (2.12%) may be used to find code which duplicates other functionality. Most of the code clone defects resulted from re-implementing functionality found in parts of the program with which the contributor was less familiar. Others were introduced by repeatedly implementing short code fragments within the same pull request (Ex. 13).

> Example 13. "... Because `parties.find(p => p.id === candidate.partyId)!.name` is now used 5 times across 3 files, ... should be abstracted in some way?"
>
> If a code fragment is repeatedly duplicated, then it should be abstracted into a method.

6.8 Dead Code Detectors

Dead Code Detectors (2.04%) might be used to detect code which is never executed. Dead code was often caused by unawareness by the developer of the potential states of the application (Ex. 14). Other causes of dead code included code migration and refactoring of functionality.

> Example 14. "Please remove this unused click event listener and the `discordLabelText_Click` function."
>
> If a method is not called, then it should be removed.

6.9 Syntax and Semantic Analyzers

Syntax and Semantic Analyzers (1.51%) may be able to detect defects such as incorrect import statements, missing or extra characters in code, and typos in identifiers (Ex. 15).

> Example 15. "Typo (missing 't'): `defaultDotsLoader`"
>
> If an identifier is referenced, then it should be defined.

6.10 String Literal Checkers

String Literal Checkers (1.51%) may be able to detect defects in string literals that lead to incorrect implementation of requirements or reduce code maintainability. String Literal Checkers may be used to detect incorrect string literals violating system requirements and altering system behavior (50.00% of defects detectable by String Literal Checkers). For

example, String Literal Checkers can check whether a correct host name is used for client-server applications. They can also check whether strings literals assigned to variables are among approved lists of strings (Ex. 16).

> Example 16. "[The value of an object property] must be one of those values [defined in a list]"
>
> If a property belongs to a specific object type, then its value must be selected from a predefined list.

String Literal Checkers might also check custom maintainability rules where specified formats are defined for string literals (50.00% of defects detectable by String Literal Checkers). For example, they can check whether log messages are printed consistently (Ex. 17).

> Example 17. "Follow the common pattern for log messages, where the first parameters are the simulation time and class name, such as the example below: `LOGGER.debug("{}: {}: {} (PEs: {}) mapped to {} (available PEs: {})",}`"
>
> If a string literal is a logger message, then it should follow the predefined format.

6.11 Code Smell Detectors

Code Smell Detectors might potentially detect 0.76% of all defects, such as code which is unnecessarily long or complex (Ex. 18).

> Example 17. "Follow the common pattern for log messages, where the first parameters are the simulation time and class name, such as the example below: `LOGGER.debug("{}: {}: {} (PEs: {}) mapped to {} (available PEs: {})",}`"
>
> If a string literal is a logger message, then it should follow the predefined format.

6.12 Memory Leak Detectors

We found that Memory Leak Detectors have the potential to detect the fewest of the defects found through code review (0.30% of all defects). These tools can detect defects like unfreed memory allocations (Ex. 19). There are several reasons why memory leak defects were infrequently observed. One may be support by modern programming languages in automatically managing memory has resulted in few defects. Another may be that these defects are simply rarely found through code review, instead surfacing only through testing.

> Example 19. "This [memory allocation] isn't being free'd.
>
> If memory is allocated, then it should eventually be freed.

RQ1 12 types of SATs may be able to detect 76% of defects identified in code reviews using the formal channel. Style Checkers and AST Pattern Checkers are most broadly applicable, potentially able to detect 33.60% and 33.70% of defects.

7 RQ2: What Features in SATs are Necessary to Detect Defects?

To find the required features in SATs to detect defects, we looked at defects detectable by SATs and identified important required features for each type of SAT.

7.1 AST Pattern Checkers

The most broadly applicable tools were AST Pattern Checkers: 25.70% of all defects. Many of these defects involved violations of project-specific rules (Ex. 20). To detect AST-based project-specific defects, AST Pattern Checkers should enable developers to author custom AST-based rules, as in extensible checkers such as PMD and RulePad (Mehrpour et al. 2020).

> Example 20. "DO NOT read information other than canvas type in this view, just get `graph-type` from the `request.GET`..."
>
> If a method is a getter reading data from an API, then it should only read the data specified in the getter identifier.

While most defects concerned a single element (e.g., block, method, or file), some involved rules crosscutting several elements. For example, one reviewer suggested that to define new REST URL endpoints through a specific REST framework, overriding superclass methods is enough and there is no need to assign annotations to the methods (Ex. 21). Tools can identify violations of this rule by checking the AST of classes and methods. To detect crosscutting and more complex AST-based defects, AST Pattern Checkers should support complex and crosscutting rules.

> Example 21. "Since this class is inherited from `viewsets.ViewSet`, you can just override `def get(self, request)`, then you don't have to add extra decorators."
>
> If a method of the target class has specific annotations and the superclass has a method with the same annotations, then the method can override the one in the superclass and remove extra annotations.

To identify necessary features for checking AST-based rules, we analyzed the defects potentially detectable by AST Pattern Checkers in detail. AST-based rules can be formulated in two parts: *when* the rule should apply (the quantifier of the rule) and *how* the rule should apply (the constraints of the rule). A defect occurs when a snippet of code satisfies the rule quantifier and violates the rule constraints. Tools such as ActiveDocumentation (Mehrpour et al. 2019) identify code snippets where only the quantifier applies as well as where both the quantifier and the constraints apply and compares the results to identify defects. More traditional defect detectors such as PMD and FindBugs detect defects by checking for the co-occurrence of both parts of the rule. For example, the rule "UseIndexOfChar" of PMD states that when the index of a single character of a string is looked for, then it is more efficient to use `String.indexOf(char)`.[2] PMD checks this rule by checking the called methods (`indexOf` and `lastIndexOf`) and the type of input argument (String or Char) and throws errors if it detects a violation.[3]

[2] https://pmd.github.io/latest/pmd_rules_java_performance.html#useindexofchar

[3] https://github.com/pmd/pmd/blob/master/pmd-java/src/main/java/net/sourceforge/pmd/lang/java/rule/performance/UseIndexOfCharRule.java

Based on the characteristics of the rule constraints we observed in our dataset, we identified three categories of AST-based rule violations: Incorrect (45.29% of defects potentially detectable by AST Pattern Checkers), Extra (25.88%), and Missing Code (28.82%) violations. An *Incorrect* Code violation occur when a code snippet follows the quantifier conditions but the implemented constraints contain incorrect code (Ex. 22). An *Extra* Code violations occur when a code snippet follows the quantifier conditions but the implemented code contains extra unwanted code (Ex. 20). A *Missing* Code violation occur when a code snippet follows the quantifier constraints but the implemented constraints are incomplete (Ex. 3).

Example 22. "Do not return status code ... either success or throw error so that client can react in a proper way."

If a method is making a request to an external server, it should process the response status and return success or throw error.

To check for these three categories of rule violations, tools should support multi-step rule verification. In the first step, tools should find the code snippets on which rules are applied by checking code against the conditions of rule quantifiers. In the next step, tools should verify the constraints of the rules on the code snippets found in the previous steps. And finally, tools should report code snippets violating the constraints. For some AST-based rules, the quantifier and constraints are applied on the same part of the AST. To check these rules, existing tools such as PMD or FindBugs are capable of executing a single AST query to check each rule (Fig. 5). However, more complex and crosscutting rules may require executing multiple AST queries.

7.2 Style Checkers

Style Checkers were the second most broadly applicable tools, potentially detecting 25.62% of all defects in our dataset. Almost half of these related to rules about the format of code, including rules about spacing, indentation, and empty lines. Less common defects related to typos in comments, naming conventions such as camel-case or lowercase letters, and program language conventions such as lambdas.

```
<ruleset name="Custom Rules"
        xmlns="http://pmd.sourceforge.net/ruleset/2.0.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://pmd.sourceforge.net/ruleset/2.0.0 https://pmd.sourceforge.io/ruleset_2_0_0.xsd">
    <description>
        Custom rules for PMD
    </description>
    <rule name="BankApplication_methods"
          language="java"
          message="BankApplication must contain the following five methods: store, update, deposit, withdraw, and destroy"
          class="net.sourceforge.pmd.lang.rule.XPathRule">
        <description>
            The void functionalities in the application should be limited to a required bank functions,
            i.e., store, update, deposit, withdraw, and destroy.
        </description>
        <priority>3</priority>
        <properties>
            <property name="xpath">
                <value>
                    <![CDATA[
                    //ClassOrInterfaceBodyDeclaration/MethodDeclaration[ResultType/@Void="true" and MethodDeclarator[not(@Image="store"
                    or @Image="update" or @Image="deposite" or @Image="withdraw" or @Image="destroy")]]
                    ]]>
                </value>
            </property>
        </properties>
    </rule>
</ruleset>
```

Fig. 5 In PMD, AST-based rules can be configured as queries of unwanted code snippets on the AST of code

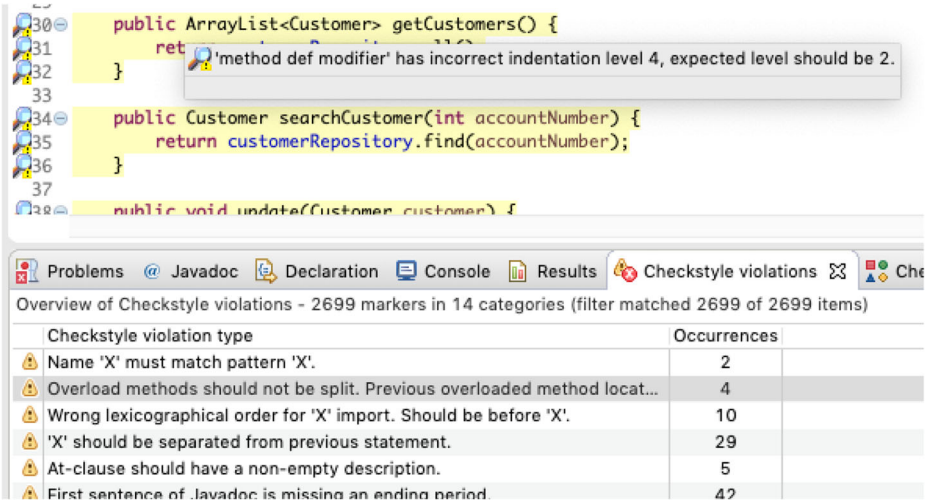

Fig. 6 CheckStyle checks sets of universal coding style rules on code, such as format of code or general naming conventions

Many defects were violations of common best practices, such as defects related to code formatting (Ex. 23 to 25), typos in comments (Ex. 26), or naming conventions (Ex. 27). Others were violations of project-specific rules defined by developers, such as imports with wildcards (Ex. 30), incorrectly structured comment blocks (Ex. 31), or missing required prefixes for method names (Ex. 32). Many of these defects are detectable by simple local checks on code as is done in CheckStyle (CheckStyle 2004) (Fig. 6).

Some defects require more complex rules to detect. For example, some projects may declare a style rule that commented code should instead be removed (Ex. 29). Syntactically valid commented code from other comments can be distinguished by simple checks,[4] but detecting unparsable commented code requires further computation and a more complex parser. Other defects concerned how code elements are ordered and grouped, such as organizing internal and external library imports (Ex. 33) or sorting methods to match the order in which they are invoked (Ex. 34). To detect ordering defects, Style Checkers might use both static analysis as well dynamic analysis to determine the order of elements (Table 3).

7.3 Less Frequent Tools

Defects potentially detectable by Continuous Integration Tools encompassed 5.06% of all defects. Most were violations of universal rules, such as outdated or inconsistent versions of libraries, missing scripts when adding or using a library or a framework, incorrect paths, or an incorrect config file format. Others were violations of project-specific rules, such as extra scripts (`pwd` or `ls` commands), required scripts for failed steps, and required libraries. Many of these might be detected by static checks.

Static Execution Simulators might potentially detect 4.91% of all defects in our dataset. By tracking data flow, these tools may help in identifying the mutability of variables (Ex. 8), confirming the need for variables (Ex. 9), and detecting possible overflow or underflow

[4] https://github.com/checkstyle/checkstyle/issues/2982

Table 3 Style Checkers are able to detect violations of common best practices as well as violations of project-specific rules. We included examples from our dataset for each category of rules

Common Best Practices	
Format of the code	Example 23. "[Add] space after while."
	Example 24. "Please keep two tabs indentation."
	Example 25. "[Add] two empty lines between classes and functions."
Typos in comments	Example 26. "start comment from capital letter as other comments."
General naming convention	Example 27. "Method name is camel case, so `getArticles` instead of `getarticles`."
Program language convention	Example 28. "[In python] a class without ancestors doesn't need a pair of parenthesis [after its identifier name]."
Commented code	Example 29. "Please remove the commented out code."
Project-Specific Rules	
Imports with wildcards	Example 30. "Don't do `import *` if it's not necessary."
Comment block structure	Example 31. "I think our convention should be to not use in-line comments, would rather it was on top of the code block its trying to explain."
Method name conventions	Example 32. "Name test methods like: testCrateTransaction, testUpdateTransaction, testUpdateTransaction_invalidId, testCreateTransaction_missingTaskId, etc..."
Ordering and organizing elements	Example 33. "Please separate the std imports into their own group."
	Example 34. "Can you please move [method] `content` after `rawContent`? In the App `rawContent` is called first."

(Ex. 35). Some defects also required the ability to detect undesired sequences of events (Ex. 7).

> Example 35. "[The variable] `overrun` can underflow in the minus operation. make it `int` here to fix."
> If a variable is mutable, then its type should be adjusted to avoid underflow.

Architectural Style Checkers might potentially identify 4.54% of all defects in our dataset. These defects were mostly project-specific. Tools might check for violations of naming conventions (both universal and project-specific), the correct location of code elements (e.g., classes, methods, etc.), the correct visibility of code elements, and the correct location of functionality within elements (Ex. 36).

> Example 36. "Instead of extending the parent class, implement the functionality as delegation pattern so we can reuse it for other callbacks if needed."
> If a class contains utility functionalities, then they should be extracted and implemented as a separate class extending a dedicated superclass.

Test Suite Quality Checker might potentially detect 2.19% of all defects in our dataset. The majority of concern the coverage of newly implemented functionalities in tests. Code coverage in tests can be checked by dynamic analysis as it is already covered in existing off-the-shelf tools like JCov (2014). Other required features include parameterizing the tests, verifying the correctness of tests (e.g., always passed or failed tests), and identifying test cases testing the same condition.

Other types of SATs are well-defined in the literature, with well-known necessary features. For example, Code Clone Detectors and Dead Code Detectors might potentially detect 2.12% and 2.04% of all defects. To detect these defects, these tools should be able to detect unreachable or unexecuted code as well as syntactic or semantic code clones through static or dynamic analysis. Syntax and Semantic Analyzers might potentially detect 1.51% of all defects by performing static analysis. String Literal Checkers might potentially detect 1.51% of all defects by identifying the String locations by static analysis and applying pattern checking techniques such as Regex to find violations. Code Smell Detectors and Memory Leak Detectors might potentially detect the least frequent defects (0.76% and 0.30% of all defects) by applying static or dynamic analysis.

RQ2 Many defects are violations of project-specific rules, which requires defect detectors to be extensible to extend the default universal checks offered. For Style Checkers, while most rules are checkable by simple local analysis, other rules such as ordering require more complex parsers or applying dynamic analysis. To detect AST-based rule violations, in addition to support project-specific rules, tools should support crafting and checking crosscutting rules through executing multiple AST queries.

8 RQ3: Which Defects Cannot be Detected by Only Using the Formal Channel?

In order to detect defects, SATs traditionally rely exclusively on interpreting code based on its syntax and semantics. Throughout the paper, we refer to tools which rely on this information about code, such as traditional SATs, as making use of the formal channel. However, in some cases, this information may be insufficient to determine the intended behavior of code. During code review, human developers may make use of additional information to identify defects. For example, a developer might read the comment of a method and, based on this, decide that the code does not correctly implement this behavior. To understand the nature of defects not detectable through the use of the formal channel alone, as used by traditional SATs, we examined defects which require more information to detect, which we label as 'defects not detectable by using only the formal channel' *NFC* (Section 4.1). We specifically examined the types of human judgments necessary to interpret these defects and the information in the natural language channel of code such as comments and identifiers (Casalnuovo et al. 2020) or in artifacts other than code that supports this.

An NFC defect may require information from various artifacts or even tacit knowledge known only by the developer themselves to be detected. For example, a developer may identify incorrect logic implemented in a method by looking at the method identifier and comments which are in the natural language channel of code (Ex. 49). Or a developer might identify an incorrect constant identifier by inspecting the use cases of the constant (the formal channel) and then comparing this to comments and identifiers describing the intent of

Table 4 Percentage of code review defects not detectable by SATs through the formal channel (NFC) by defect type. The *Count* and *Overall %* Columns are defined as in Table 2

Defect type	All Defects		NFC Defects
	Count	Overall %	Overall %
All NFC	314	23.73 %	100 %
Maintainability	154	11.64 %	49.04 %
Implementation	88	6.65 %	28.03 %
Requirement	30	2.27 %	9.55 %
Other	42	3.17 %	13.38 %

the constant (the natural language channel) (Ex. 37). A developer may also use information beyond the code to detect a defect. For example, a developer might choose between several possible ways of logging an event by considering the meaning and importance of the event, which cannot be determined from the formal channel of code (Ex. 42). This information may be instead be described in documentation or rely on information about the project tacitly held by the developer.

Investigating defects not detectable through the formal channel alone is helpful to investigate the potential for future tools which use information beyond the formal channel in their analysis of code. Through the use of ML or NLP techniques, future tools might perform some of the same analysis as a human developer, inferring intent from the natural language channel information found in code identifiers or comments or through other artifacts.

We found that NFC defects involve maintainability defects such as incorrectly named identifiers (Ex. 37), implementation defects like incorrect implementations (Ex. 41), and requirement defects like incomplete implementations (Ex. 43). We found that, among all NFC defects, maintainability defects are the most frequent, encompassing 49.04% of all NFC defects, followed by implementation defects (28.03% of NFC defects). Our results are summarized in Table 4.

8.1 Maintainability

The most frequent type of NFC defects were maintainability defects (49.04% of NFC defects). Many concerned identifier naming. Compared to issues such as camel-case letters in method names (Ex. 27), which might be detected by Style Checkers, these defects did not involve a list of rules which could be easily enumerated and instead required human judgment and common-sense knowledge. For example, a developer may view an identifier as not correctly communicating the purpose of a variable.

> Example 37. "I'd rename [`CRSF_LINK_STATUS_UPDATE_US`] to `CRSF_LINK_STATUS_UPDATE_TIMEOUT_US` or similar to indicate that this is a timeout value."
> If a constant needs to be defined, then its name should reflect its purpose.

Another common issue was in the use of comments, including comments which were missing, unnecessary, incorrect, incomplete, or misplaced. For example, one defect concerned a missing inline comment. This was not universally prohibited or required, but

necessary only in cases where code was hard to understand. This requires developer judgement to ascertain.

> Example 38. "A comment wouldn't hurt here that tells that its basically looping as long the result is `WSAEFAULT` because that is the result in case the buffer is too small."
> If the logic of code is not easy to understand, then a comment is required.

Others issues concerned the readability of code, particularly those that dealt with its organization and how this communicated intent and whether code constituted one intent or multiple intents.

> Example 39. "These are two separate queries anyway, so there's no point in doing it in one LINQ statement and hiding it in a `gameEventAndGames`. Just do two LINQs assigning to `gameEvent` and `games`."
> If conceptually there are multiple queries, then write each query separately.

8.2 Implementation

The second most common category of NFC defects involved incorrect implementations (28.03% of NFC defects). These involved identifying incomplete, unnecessarily complex, or incorrect implementations. For example, one reviewer identified a defect where they found that a specific value should never be passed as an argument but should instead be computed from a different argument (Ex. 40).

> Example 40. "I think passing the cache as a ... third argument to [the function] and fetching the solution inside that function might work without the overloader."
> If implementing a method which requires a specific value, then the value can be accessed by passing a different value as an argument and calculating the desired value within the method from the argument.

Other implementations were incomplete, and reviewers suggested additional work to be done based on their understanding of what functionality was needed. One reviewer found that an error was particularly important, and required a new API error to signal it (Ex. 41).

> Example 41. "This error condition means that we have run out of guest PCI slots ... So we should introduce a new API error for it."
> If an error is important, then there should be a specific API error dedicated for it.

Identifying this defect requires evaluating the meaning and significance of errors which occur.

Other implementations were incorrect. One reviewer identified an event that was logged at an incorrect level, based on their understanding of the importance of the data to the user (Ex. 42).

Example 42. "[`LOG.info()`] should probably become `LOG.debug()`, given [the output] when this is set to [`LOG.info()`]."

If the output information is not useful to the end user, then the `debug` logger should be used.

Identifying this defect requires making a human judgement about the importance of an event to the user.

8.3 Requirements

9.55% of NFC defects concerned incorrect implementation of requirements. These included implementations that were incorrect, incomplete, or which violated constraints. For example, one reviewer felt that checks which signaled failures too frequently should be avoided when possible, with the condition logged rather than signaling a failure (Ex. 43).

Example 43. "This check, while correct, may be a bit aggressive in the current network and might result in a lot of failures ... We can still enforce that we don't announce by simply not sending [signatures] ourselves. Please don't fail the channel over such a minor offence, but log it and we can tighten this later."

If a check leads to excessive failures, then it should be avoided.

Other requirements defects concerned incomplete implementations which required additional code. Viewing it as necessary that when important events occur, a user should be notified, one reviewer found a case where this did not occur (Ex. 44).

Example 44. "It might be useful to add a message here explaining that the effect has timed out for [the end users]."

If an event is important to end users, then they should receive a notification when it occurs.

Requirements defects also concerned constants which were incorrectly chosen. One reviewer felt that a constant allocating resources was incorrectly chosen, in light of the amount of resources which they expected might be required (Ex. 45).

Example 45. "How did you come up with this number 100k [for the stack size]? Seems low."

If a resource limit is specified, it should be high enough to support typical use in practice.

8.4 Less Frequent Defect Types

User interface defects (5.73% of NFC defects) involved incomplete, incorrect, or undesired implementations of user interface features. Defects involved an incorrect implementation of a user interface element (Ex. 47) or an incorrect color scheme for user interface elements (Ex. 48). For example, one reviewer felt that, whenever the user rejected an action, they should be given a timer to revert it before it is finalized (Ex. 46).

Example 46. "This should decrease influence, and add a 'refused to remove troops' countdown - so this doesn't pop up every single turn"

If an end user rejects an action, then the application should display a timer to allow the user to revert the action before finalizing it.

Example 47. "Let's not do `variant="h6"` for this label. We want the hierarchy to emphasize the title on the left. Let's just go with the default typography variant (no variant prop needed)."

If the visual properties of an element distort the emphasis of the title, then its properties should be left as default.

Example 48. "I suggest that you choose colors which would be easier to see when playing the game."

If colors are used for visual elements, then they should be easy to distinguish for end users.

We found that 4.46% of NFC defects concerned incorrect, missing, and hard-to-read tests. Defects involved issues such as test suites which had not been updated to match the implementation (Ex. 49), incorrect use of boundary values (Ex. 50), unreadable tests (Ex. 51), and additional needed tests (Ex. 52).

Example 49. "If this now relies on the object keys order, this test should be updated to reflect that."

If a test is designed for a specific feature in the test, then its implementation and its title should reflect that.

Example 50. "(optional) As for the chosen value, it is better to choose the "edge value", i.e. the first one that would be incorrect (-2). If we choose -4, we aren't really covering -3 and -2 with this test."

If a value needs to be assigned in a test, then it should be an edge value.

Example 51. "I'd prefer for this function to be parameterized, so `X-Foo`, `X-Bar` could be in (or near) the `TestFetchSignWithForwardedRequestHeaders` function itself. This would make it easier to compare the test fixture against the expected value."

If an assertion in a test contains several important input parameters, then it should be parameterized.

Example 52. "Please add a test that returns actual numbers that you expect to get when you send some value. like sending (winner=p1, p1=1400, p2=1400) returns (p1=1450, p2=1350)."

Build configuration defects encompassed 1.91% of NFC defects in our dataset and concerned incorrect or incomplete configuration code. One reviewer had a rule that a specific configuration file was only expected to be listing the tests that were expected to fail (Ex. 53).

Example 53. "Please remove this particular test result [from the config file], since it passes with your changes."

If a test is expected to `PASS`, then it should not be listed in a specific configuration file.

The least frequent category of NFC defects were performance defects (1.27%). For example, one reviewer identified a case where a performance optimization was possible, and a slow method called many times might be called just once. (Ex. 54).

Example 54. "(optional) If we want to we could reduce the multiple traversal issue by calling `preOrder` just once from within `validateOptions()`."

If a method has a high execution cost, then it should be called as few times as possible.

RQ3 We found that 23.73% of all defects identified by code reviews are not detectable by existing types of SATs using the formal channel. NFC defects involve violations of rules which require human judgment to check. Almost half of these were maintainability defects, with issues involving identifier naming and the use of comments. Other common defects included implementation defects involving incorrect, overly complex, or incomplete code and requirement defects involving requirements which were incorrectly implemented.

9 Threats to Validity

Our study has several important limitations and internal and external threats to validity.

Internal Validity A potential threat to the internal validity of our results is the accuracy of our coding. To create our coding scheme, we first applied two cycles of open coding, creating a definition for each code. To ensure our coding scheme could be accurately and repeatably applied, we applied and refined the definitions, clarifying ambiguity across multiple cycles of closed coding. To label each code review comment, we needed to understand reviewers' intent as well as the context of the review comment. To gain insight into the review comments, we used all available artifacts included and referenced by the pull requests. However, the complexity of the projects, and the unavailability of some artifacts, sometimes made this challenging. It was also necessary for us to interpret the intent of each comment. To do this accurately, we consulted previous and subsequent review comments in the same pull request to add additional context. Reviewers may also express multiple defects stated in a single review comment. In our coding process, we selected the most emphasized defect in each comment.

External Validity Our dataset of PR comments was taken exclusively from open source projects. Our findings might potentially differ for commercial projects. However, our dataset is diverse in the types of the projects, number of contributors, and programming languages.

The repositories in the dataset range from group projects with a small number of contributors to open source projects with many contributions. A limitation of our study comes from focusing on defects found through code review. It is possible that some defects are simply never, or only very rarely, found by code review, but are instead revealed only through testing or other quality assurance measures. Further work is needed to determine how the types of defects found in code review, which we examine, are similar or different from those found by other quality assurance measures.

10 Discussion

In this paper, we investigated the potential of SATs to find more defects. We found that SATs have the potential to detect 76% of defects identified in code reviews, considerably more than the 4.5% of defects found to be detectable by Error Prone, Infer, and Spotbugs (Habib and Pradel 2018) or 16% of code review defects found by PMD (Singh et al. 2017). There are several potential interpretations of this divergence. Defects may have occurred in code written in programming languages not supported by these tools. The defects might be able to be detected by these tools, but require additional rules which had not been written to detect. Finally, some of the defects might, in principle, be found through the use of these techniques, but might require more powerful static analysis techniques with greater precision than is possible today.

10.1 Analyzing Code Review Defects

Code review defects have long been used as a proxy to evaluate the ability of defect detectors to find defects (Singh et al. 2017; Wagner et al. 2005; Beller et al. 2016). The goal of a defect detector must be to check for defects as defined by a project's developers. While developers reviewing code may have different definitions of correctness (Sadowski et al. 2018a) or even suggest changes that might seemingly be viewed as unnecessary or inconsistent, we believe that measuring the success of a tool against what its users, the developers, wish it might find is still the most important metric with which to define its success.

When we marked defects as being potentially detectable by static analysis techniques, this does not imply that this is possible with today's static analysis tools. Our analysis is aimed at motivating future tools, and is consequently inherently optimistic in considering what might be possible. In practice, some of these defects may require much more powerful analysis techniques, or might, in fact, require analysis techniques that are impossible to create.

10.2 Using SATs to Detect More Defects

Many of the rules we identified were project-specific, necessitating developers in specific projects to write them rather than relying on the creators of the SAT to have already written them. Whereas today's tools are known for general rules but are extensible to project-specific rules (e.g., PMD, Tricorder), our results strongly suggest that future defect detectors would greatly increase their ability to identify defects by embracing project-specific rules. Making it easier for developers to themselves quickly author project-specific rules might help bridge this gap. Our findings can guide practitioners to better understand and prioritize different types of SATs.

We found AST Pattern Checkers to have wide applicability, able to potentially detect 25% of all code review defects. This is again higher than the 16% of code review defects that have been identified to be detectable found by PMD, a common AST Pattern Checker (Singh et al. 2017). As Static Analysis Tools, AST Patterns Checkers are comparatively simple, relying on syntactic checks rather than tracking data flow or more complex rules. Our results suggest several important ways that these tools might be extended to identify more defects. We found three types of AST-based violations detectable by AST Pattern Checkers, including Incorrect Code violations, which are not directly supported by existing SATs. This suggests the value of investing more in further improving AST Pattern Checkers, where the important barriers remain primarily in what rules are supported and the difficulty of supporting individual developers in authoring project-specific rules.

Beyond simply identifying the presence of a defect, developers also increasingly expect Static Analysis Tools to help in the process of proposing a fix (Johnson et al. 2013). While current SATs can detect 2 out of the 3 types of AST-based rule violations we identified, many are not yet capable of proposing fixes. To propose fixes to Missing or Extra Code violations, tools may suggest code by using separate AST patterns for a quantifier (when the rule applies) and a constraint (what the code must satisfy). For example, ActiveDocumentation (Mehrpour et al. 2019) compares code against two separate AST patterns, enabling it to identify Missing Code violations. PMD (Copeland 2005) and FindBugs (Hovemeyer and Pugh 2004) can detect undesired code snippets, and thus, may detect Missing or Extra Code violations. However, they are not able to differentiate Incorrect code and other types of violations. In addition, they are not always able to suggest fixes.[5]

SATs such as Memory Leak Detectors that only detect a small percentage of the defects found in code review, might still be valuable to developers in some cases. Deciding on using a SAT requires the developers to make an engineering trade-off to balance the cost of running the tool and against the potential benefits of using the tool. In some cases, even infrequent defects may be severe enough that they warrant substantial effort to prevent.

In this study, we used a new qualitative method to investigate the potential of tools to identify defects. Specifically, we focused on evaluating the potential of static analysis tools to detect defects found in code reviews. While static analysis tools are widely used to detect defects, many non-static analysis techniques, such as mutation testing or ML-based techniques, have also increased in popularity in recent years. Comparing the ability of static and non-static analysis techniques would help weigh the cost of applying non-SAT defect detection techniques against the benefits gained through the defects found by each analysis method. Future studies might make use of our qualitative method to examine other types of analysis techniques such as unit tests and compare the types and quantity of defects potentially detectable by a variety of analysis techniques.

10.3 Detecting NFC Defects

We found that 24% of code review defects cannot be detected by existing types of SATs, as these defects require human judgment to identify and information beyond the formal channel information that traditional SATs use to identify defects. However, these defects may not be entirely impossible for tools to detect. Tools which employ techniques from

[5]E.g., some tools offer quick fixes for limited types of defects found by FindBugs, https://github.com/kjlubick/fb-contrib-eclipse-quick-fixes

machine learning and natural language processing may make it possible to simulate the process human developers use to make judgements about the correctness of code. Recent work has begun to explore the potential for tools to utilize machine learning techniques to detect defects in code (e.g., Nam and Kim 2015; Huo et al. 2018). ML-based defect detectors use information other than the formal channel of code such as comments (Huo et al. 2018), identifier names (Pradel and Sen 2018), or previous defect fixes (Wang et al. 2020) to predict defects offering optimism to detect NFC defects. Natural language processing techniques may also be used to process source code information and detect NFC defects such as input parameter misplacement(Rice et al. 2017). Recent progress in designing deep learning-based tools, such as GitHub CoPilot (GitHub 2021), demonstrate the great potential of the natural language channel in working with code.

10.4 Documentation and SATs

Our findings also suggest the potential for a closer relationship between documentation and SATs. A common challenge developers face is the inaccessibility and incompleteness of information in code. Code review is an important approach for on-boarding and helping developers gain knowledge about code. In many cases, this knowledge is not explicitly structured or documented. Source code knowledge is often written down by developers only in code reviews, leaving it scattered in pull requests and buried in review comments. Rather than simply address issues in each individual commit, developers who find issues in code review might instead write checkable rules to find future issues of the same form. Tools such as Getafix (Bader et al. 2019) have already begun to explore the potential for this interaction paradigm. Our work suggests the broad potential for a variety of types of SATs to be constantly used by developers to create rules which capture important project-specific knowledge.

10.5 Future of Code Review

In current practice, the code review process has little to no connection with defect detectors, as code reviewers examine the submitted code manually to identify defects. In some software development teams, developers work to write style rules and ask contributors to check their code against them before submitting the code. Our results suggest that many of the issues developers are finding during code review might be written as rules at the time developers perform code review. For example, a developer finding an issue during code review might see an issue in one place, write a new rule to check this, and then, as part of their code review, detect other violations of the rule. The code review might then include a new rule and highlight current violations of the rule. Our results suggest a potential for code review tools to more directly integrate rule checkers, in particular supporting easy authoring of new rules. This would save code reviewers from repeatedly applying rules to code, pushing this work onto tools, making rules more consistently checked, potentially decreasing the number of defects in code, and reducing the cost of code review.

Funding This work was supported in part by the National Science Foundation under grant NSF CCF-1703734.

Data Availability Our dataset is publicly available on https://doi.org/10.6084/m9.figshare.14925222

Declarations

Conflict of Interest The authors have no financial or non-financial interest.

References

Aftandilian E, Sauciuc R, Priya S, Krishnan S (2012) Building useful program analysis tools using an extensible java compiler. In: International working conference on source code analysis and manipulation (SCAM), pp 14–23. https://doi.org/10.1109/SCAM.2012.28

Aghajani E, Nagy C, Vega-Márquez OL, Linares-Vásquez M, Moreno L, Bavota G, Lanza M (2019) Software documentation issues unveiled. In: International conference on software engineering (ICSE), pp 1199–1210. https://doi.org/10.1109/icse.2019.00122

Aldrich J, Chambers C, Notkin D (2002) Archjava: connecting software architecture to implementation. In: International conference on software engineering (ICSE), pp 187–197. https://doi.org/10.1145/581339.581365

Ayewah N, Pugh W (2010) The google findbugs fixit. In: International symposium on software testing and analysis (ISSTA), pp 241–252. https://doi.org/10.1145/1831708.1831738

Bacchelli A, Bird C (2013) Expectations, outcomes, and challenges of modern code review. In: International conference on software engineering (ICSE), pp 712–721. https://doi.org/10.1109/icse.2013.6606617

Bader J, Scott A, Pradel M, Chandra S (2019) Getafix: learning to fix bugs automatically. In: Conference on object-oriented programming systems, languages and applications (OOPSLA), pp 159:1–159:27. https://doi.org/10.1145/3360585

Balachandran V (2013) Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation. In: International conference on software engineering (ICSE), pp 931–940. https://doi.org/10.1109/icse.2013.6606642

Barik T, Ford D, Murphy-Hill E, Parnin C (2018) How should compilers explain problems to developers? In: European software engineering conference and international symposium on the foundations of software engineering (ESEC/FSE), pp 633–643. https://doi.org/10.1145/3236024.3236040

Basili VR, Selby RW (1987) Comparing the effectiveness of software testing strategies. Trans Softw Eng SE-13(12):1278–1296. https://doi.org/10.1109/tse.1987.232881

Beller M, Bacchelli A, Zaidman A, Juergens E (2014) Modern code reviews in open-source projects: which problems do they fix? In: International conference on mining software repositories (MSR), pp 202–211. https://doi.org/10.1145/2597073.2597082

Beller M, Bholanath R, McIntosh S, Zaidman A (2016) Analyzing the state of static analysis: a large-scale evaluation in open source software. In: International conference on software analysis, evolution, and reengineering (SANER), pp 470–481. https://doi.org/10.1109/saner.2016.105

Bessey A, Block K, Chelf B, Chou A, Fulton B, Hallem S, Gros C, Kamsky A, McPeak S, Engler DR (2010) A few billion lines of code later: using static analysis to find bugs in the real world. Commun ACM 53(2):66–75. https://doi.org/10.1145/1646353.1646374

Beyer D, Keremoglu ME (2011) Cpachecker: a tool for configurable software verification. In: International conference on computer aided verification, pp 184–190. https://doi.org/10.1007/978-3-642-22110-1_16

Board I (1993) IEEE Standard classification for software anomalies. IEEE Std, pp 1044

Brunet J, Murphy GC, Terra R, Figueiredo J, Serey D (2014) Do developers discuss design? In: International conference on mining software repositories (MSR), pp 340–343. https://doi.org/10.1145/2597073.2597115

Burnstein I (2002) Practical software testing. Springer, Berlin

Calcagno C, Distefano D, Dubreil J, Gabi D, Hooimeijer P, Luca M, O'Hearn PW, Papakonstantinou I, Purbrick J, Rodriguez D (2015) Moving fast with software verification. In: NASA formal methods symposium, Lecture Notes in Computer Science, vol 9058, pp 3–11. https://doi.org/10.1007/978-3-319-17524-9_1

Casalnuovo C, Barr ET, Dash SK, Devanbu P, Morgan E (2020) A theory of dual channel constraints. In: International conference on software engineering, new ideas and emerging results (ICSE-NIER), pp 25–28. https://doi.org/10.1145/3377816.3381720

CheckStyle (2004) http://checkstyle.sourceforge.net

Chillarege R, Bhandari IS, Chaar JK, Halliday MJ, Moebus DS, Ray BK, Wong MY (1992) Orthogonal defect classification-a concept for in-process measurements. Trans Softw Eng 18(11):943–956. https://doi.org/10.1109/32.177364

Christakis M, Bird C (2016) What developers want and need from program analysis: an empirical study. In: International conference on automated software engineering (ASE), pp 332–343. https://doi.org/10.1145/2970276.2970347

Copeland T (2005) PMD applied. Centennial Books

Crockford D (2011) Jslint: the javascript code quality tool. http://www.jslint.com

Ebert F, Castor F, Novielli N, Serebrenik A (2018) Communicative intention in code review questions. In: International conference on software maintenance and evolution (ICSME), pp 519–523. https://doi.org/10.1109/icsme.2018.00061

Fard AM, Mesbah A (2013) JSNOSE: detecting Javascript code smells. In: International working conference on source code analysis and manipulation (SCAM), pp 116–125. https://doi.org/10.1109/scam.2013.6648192

Ghorbani N, Garcia J, Malek S (2019) Detection and repair of architectural inconsistencies in java. In: International conference on software engineering (ICSE), pp 560–571. https://doi.org/10.1109/icse.2019.00067

Gilb T, Graham D, Finzi S (1993) Software inspection. Addison-Wesley

GitHub (2021) Github copilot. https://copilot.github.com/

Gousios G (2013) The ghtorrent dataset and tool suite. In: International conference on mining software repositories (MSR), pp 233–236. https://doi.org/10.1109/msr.2013.6624034

Group I et al (2010) 1044-2009-ieee standard classification for software anomalies. IEEE, New York

Habib A, Pradel M (2018) How many of all bugs do we find? A study of static bug detectors. In: International conference on automated software engineering (ASE), pp 317–328. https://doi.org/10.1145/3238147.3238213

Hovemeyer D, Pugh W (2004) Finding bugs is easy. In: Conference on object-oriented programming systems, languages, and applications (OOPSLA), pp 132–136. https://doi.org/10.1145/1028664.1028717

Humphrey WS (1995) A discipline for software engineering, 1st edn. Addison-Wesley

Huo X, Yang Y, Li M, Zhan D (2018) Learning semantic features for software defect prediction by code comments embedding. In: International conference on data mining (ICDM), pp 1049–1054. https://doi.org/10.1109/ICDM.2018.00133

JCov (2014) https://wiki.openjdk.java.net/display/CodeTools/jcov

Jenkins (2019) https://jenkins.io

Johnson B, Song Y, Murphy-Hill E, Bowdidge R (2013) Why don't software developers use static analysis tools to find bugs? In: International conference on software engineering (ICSE), pp 672–681. https://doi.org/10.1109/icse.2013.6606613

Kamiya T, Kusumoto S, Inoue K (2002) CCFInder: a multilinguistic token-based code clone detection system for large scale source code. Trans Softw Eng 28(7):654–670. https://doi.org/10.1109/tse.2002.1019480

Knodel J, Popescu D (2007) A comparison of static architecture compliance checking approaches. In: Working IEEE/IFIP conference on software architecture (WICSA), pp 12–12. https://doi.org/10.1109/wicsa.2007.1

Kruchten P (2004) An ontology of architectural design decisions in software-intensive systems. In: Groningen workshop on software variability management

Laitenberger O (1998) Studying the effects of code inspection and structural testing on software quality. In: International symposium on software reliability engineering (ISSER), pp 237–246. https://doi.org/10.1109/ISSRE.1998.730887

Laitenberger O, DeBaud JM (2000) An encompassing life cycle centric survey of software inspection. J Syst Softw 50(1):5–31. https://doi.org/10.1016/s0164-1212(99)00073-4

Landis JR, Koch GG (1977) The measurement of observer agreement for categorical data. Biometrics 159–174. https://doi.org/10.2307/2529310

LaToza TD, Myers BA (2011) Visualizing call graphs. In: Symposium on visual languages and human-centric computing (VL/HCC), pp 117–124. https://doi.org/10.1109/VLHCC.2011.6070388

Little G, Miller RC (2007) Keyword programming in java. In: International conference on automated software engineering (ASE), pp 84–93. https://doi.org/10.1007/s10515-008-0041-9

Mäntylä MV, Lassenius C (2009) What types of defects are really discovered in code reviews? Trans Softw Eng 35(3):430–448. https://doi.org/10.1109/tse.2008.71

Mao K, Harman M, Jia Y (2016) Sapienz: multi-objective automated testing for android applications. In: International symposium on software testing and analysis (ISSTA), pp 94–105. https://doi.org/10.1145/2931037.2931054

Mehrpour S, LaToza TD, Kindi RK (2019) Active documentation: helping developers follow design decisions. In: Symposium on visual languages and human-centric computing (VL/HCC), pp 87–96. https://doi.org/10.1109/vlhcc.2019.8818816

Mehrpour S, LaToza TD, Sarvari H (2020) Rulepad: interactive authoring of checkable design rules. In: European software engineering conference and international symposium on the foundations of software engineering (ESEC/FSE), pp 386–397. https://doi.org/10.1145/3368089.3409751

Nam J, Kim S (2015) CLAMI: defect prediction on unlabeled datasets (t). In: International conference on automated software engineering (ASE), pp 452–463. https://doi.org/10.1109/ASE.2015.56

Panichella S, Arnaoudova V, Di Penta M, Antoniol G (2015) Would static analysis tools help developers with code reviews? In: International conference on software analysis, evolution, and reengineering (SANER), pp 161–170. https://doi.org/10.1109/saner.2015.7081826

Pascarella L, Spadini D, Palomba F, Bruntink M, Bacchelli A (2018) Information needs in contemporary code review. Human-Comput Interact 2(CSCW) 27:1–135. https://doi.org/10.1145/3274404

Pradel M, Sen K (2018) Deepbugs: a learning approach to name-based bug detection. In: ACM on programming languages OOPSLA, vol 2, pp 147:1–147:25. https://doi.org/10.1145/3276517

Rice A, Aftandilian E, Jaspan C, Johnston E, Pradel M, Arroyo-Paredes Y (2017) Detecting argument selection defects. In: ACM On programming languages, OOPSLA, vol 1, pp 104:1–104:22. https://doi.org/10.1145/3133928

Rigby PC, Bird C (2013) Convergent contemporary software peer review practices. In: European software engineering conference and international symposium on the foundations of software engineering (ESEC/FSE), pp 202–212. https://doi.org/10.1145/2491411.2491444

Romano S, Scanniello G, Sartiani C, Risi M (2016) A graph-based approach to detect unreachable methods in java software. In: Symposium on applied computing (SAC), pp 1538–1541. https://doi.org/10.1145/2851613.2851968

Roy CK, Cordy JR (2007) A survey on software clone detection research. Tech. rep. Queen's University at Kingston, Ontario

Runeson P, Wohlin C (1998) An experimental evaluation of an experience-based capture-recapture method in software code inspections. Empir Softw Eng 3(4):381–406. https://doi.org/10.1023/A:1009728205264

Runeson P, Andersson C, Thelin T, Andrews A, Berling T (2006) What do we know about defect detection methods? IEEE Softw 23(3):82–90. https://doi.org/10.1109/MS.2006.89

Sadowski C, van Gogh J, Jaspan C, Söderberg E, Winter C (2015) Tricorder: building a program analysis ecosystem. In: International conference on software engineering (ICSE), pp 598–608. https://doi.org/10.1109/icse.2015.76

Sadowski C, Aftandilian E, Eagle A, Miller-Cushon L, Jaspan C (2018a) Lessons from building static analysis tools at Google. Commun ACM 61(4):58–66. https://doi.org/10.1145/3188720

Sadowski C, Söderberg E, Church L, Sipko M, Bacchelli A (2018b) Modern code review: a case study at google. In: International conference on software engineering: software engineering in practice (ICSE-SEIP), pp 181–190. https://doi.org/10.1145/3183519.3183525

Saldaña J (2015) The coding manual for qualitative researchers. Sage

Seaman CB (1999) Qualitative methods in empirical studies of software engineering. Trans Softw Eng 25(4):557–572

Shafiq S, Mashkoor A, Mayr-Dorn C, Egyed A (2021) A literature review of using machine learning in software development life cycle stages. IEEE Access 9:140896–140920. https://doi.org/10.1109/ACCESS.2021.3119746

Sharma T, Spinellis D (2018) A survey on software smells. J Syst Softw 138:158–173. https://doi.org/10.1016/j.jss.2017.12.034

Silva MCO, Valente MT, Terra R (2016) Does technical debt lead to the rejection of pull requests? In: Symposium on information systems on Brazilian symposium on information systems: information systems in the cloud computing era (SBSI), pp 248–254. https://doi.org/10.5555/3021955.3021997

Singh D, Sekar VR, Stolee KT, Johnson B (2017) Evaluating how static analysis tools can reduce code review effort. In: Symposium on visual languages and human-centric computing (VL/HCC), pp 101–105. https://doi.org/10.1109/vlhcc.2017.8103456

SonarSource (2022) Code quality and code security. https://www.sonarqube.org/

Structure101 (2019) https://structure101.com

Sui Y, Ye D, Xue J (2012) Static memory leak detection using full-sparse value-flow analysis. In: International symposium on software testing and analysis (ISSTA), pp 254–264. https://doi.org/10.1145/2338965.2336784

Svajlenko J, Roy CK (2015) Evaluating clone detection tools with bigclonebench. In: International conference on software maintenance and evolution (ICSME), pp 131–140. https://doi.org/10.1109/icsm.2015.7332459

Thelin T, Runeson P, Wohlin C (2003) Prioritized use cases as a vehicle for software inspections. IEEE Softw 20(4):30–33. https://doi.org/10.1109/ms.2003.1207451

Thung F, Lucia, Lo D, Jiang L, Rahman F, Devanbu PT (2012) To what extent could we detect field defects? An empirical study of false negatives in static bug finding tools. In: International conference on automated software engineering (ASE), pp 50–59. https://doi.org/10.1145/2351676.2351685

Travis CI (2019) https://travis-ci.org

Ueda Y, Ihara A, Ishio T, Matsumoto K (2018) Impact of coding style checker on code review—a case study on the openstack projects. In: International workshop on empirical software engineering in practice (IWESEP), pp 31–36. https://doi.org/10.1109/iwesep.2018.00014

van Emden E, Moonen L (2002) Java quality assurance by detecting code smells. In: Conference on reverse engineering, pp 97–106. https://doi.org/10.1109/WCRE.2002.1173068

Vassallo C, Panichella S, Palomba F, Proksch S, Zaidman A, Gall HC (2018) Context is king: the developer perspective on the usage of static analysis tools. In: International conference on software analysis, evolution, and reengineering (SANER), pp 38–49. https://doi.org/10.1109/saner.2018.8330195

Viviani G, Janik-Jones C, Famelis M, Murphy GC (2018) The structure of software design discussions. In: International workshop on cooperative and human aspects of software engineering (CHASE), pp 104–107. https://doi.org/10.1145/3195836.3195841

Wagner S, Jürjens J, Koller C, Trischberger P (2005) Comparing bug finding tools with reviews and tests. In: Testing of communicating systems, pp 40–55. https://doi.org/10.1007/11430230_4

Wang S, Liu T, Nam J, Tan L (2020) Deep semantic feature learning for software defect prediction. Trans Softw Eng 46(12):1267–1293. https://doi.org/10.1109/TSE.2018.2877612

Wiegers KE (2002) Peer reviews in software: a practical guide. Addison-Wesley Boston

Yan D, Xu G, Yang S, Rountev A (2014) Leakchecker: practical static memory leak detection for managed languages. In: International symposium on code generation and optimization, pp 87–97. https://doi.org/10.1145/2581122.2544151

Zampetti F, Scalabrino S, Oliveto R, Canfora G, Di Penta M (2017) How open source projects use static code analysis tools in continuous integration pipelines. In: International conference on mining software repositories (MSR), pp 334–344. https://doi.org/10.1109/msr.2017.2