Assert Use in GitHub Projects

Casey Casalnuovo, Prem Devanbu, Abilio Oliveira, Vladimir Filkov, Baishakhi Ray
Computer Science Dept., Univ. of California, Davis
{ccasal,ptdevanbu,vfilkov,abioliveira,bairay}@ucdavis.edu

***Abstract*—*Asserts* have long been a strongly recommended (if non-functional) adjunct to programs. They certainly don't add any user-evident feature value; and it can take quite some skill and effort to devise and add useful asserts. However, they are believed to add considerable value to the developer. Certainly, they can help with automated verification; but even in the absence of that, claimed advantages include improved understandability, maintainability, easier fault localization and diagnosis, all eventually leading to better software quality. We focus on this latter claim, and use a large dataset of asserts in C and C++ programs to explore the connection between asserts and defect occurrence. Our data suggests a connection: functions with asserts do have significantly fewer defects. This indicates that asserts do play an important role in software quality; we therefore explored further the factors that play a role in assertion placement: specifically, *process factors* (such as developer experience and ownership) and *product factors*, particularly interprocedural factors, exploring how the placement of assertions in functions are influenced by local and global network properties of the callgraph. Finally, we also conduct a differential analysis of assertion use across different application domains.**

I. Introduction

The idea of assertions, which can be automatically checked at runtime, dates back some 40 years [48]. Most popular programming languages, including `C`, `C++`, `Java`, `Python` all provide support for assertions and run-time checking; some languages even consider assertions and assertion-based programming as central to their design (*e.g.,*, `Eiffel`, `Turing`). Assertions are widely taught in undergraduate curricula, and it is reasonable to assume that most modern programmers are well-aware of their use and claimed advantages.

Run-time checking is certainly not the only use of assertions; the use of assertions in program verification also has a distinguished history, dating back to Floyd in the 1960s [18]. While the technology of automated verification has made tremendous strides, Hoare [21] laments that assertions in practice are not often used in the context of automated verification.

However, the above mentioned purpose of assertions, *viz.,* for run-time checking, is, in fact, often exploited by programmers in the trenches. Steve McConnell, the author of many popular programming "cook-books" for software practitioners, heartily advocates [30] the use of assertions to give programs a "death wish". This is in order to promote the timely and highly visible failure of a program, should it reach a state in which there are some inconsistent or invalid data values clearly in evidence. Such an immediate and self-evident failure would be far easier to diagnose than a delayed failure obscured by more widespread data corruption.

In addition to supporting debugging and fault localization, asserts are also thought to promote readability. For example, the intended purpose and function of a loop arguably becomes far more evident in the presence of an assertion indicating the termination condition, or an invariant assertion. Thus, besides aiding fault localization during intensive coding sessions, assertions may also help programmers better understand code, and avoid constructing faulty code in the first place.

So, it is reasonable to advocate for assertion use on the basis of the above two arguments: assertions will make for more informed programmers who make fewer mistakes, *and* help isolate errors quickly even if mistakes were made. All the above discussion, admittedly, is largely of a theoretical nature. Do these theories hold up in practice? The central concern of this paper is *a study of the practical use of assertions*. We collect a large corpus of the 100 most popular `C` and `C++` software projects on GitHub. Our empirical analysis of this corpus, yields the following findings:

1) Assertions are widely used in popular `C` and `C++` projects. 69 of the 100 projects contain more than a minimal presence of assertions, and this subset of projects contain in total 35M lines of code. In these projects, we find that 4.6% of methods contain assertions.
2) Adding asserts has a small (but significant) association with reduced bug density; also as the number of developers working in a function increases, adding more asserts is associated with reduced occurrence of new bugs.
3) Asserts tend to be added to functions by developers with higher ownership and experience in that function.
4) In the call graph network structure, functions with asserts are more likely to play the role of hubs, a network role that gathers and dispenses information from/to other nodes.
5) We compared the number of asserts in projects with different application domains, but found that domain did not significantly affect the number of asserts used.

II. Research Goals

We sought to understand how assertions are used in practice, in particular the process outcomes that are associated with the use (or disuse) of assertions.

We begin with the oft-stated goal of assertions, here articulated eloquently in the `Python` Wiki[1]

> *Assertions are a systematic way to check that the internal state of a program is as the programmer expected, with the goal of catching bugs. In particular, they're good for catching false assumptions that were made while writing*

[1] https://wiki.python.org/moin/UsingAssertionsEffectively

the code, or abuse of an interface by another programmer. In addition, they can act as in-line documentation to some extent, by making the programmer's assumptions obvious.

The strong implication that assertions are a way to improve quality outcomes is unmistakable. This leads directly to our first research concern:

RQ1. How does assertion use relate to defect occurrence?

In examining the last sentence of the quote above, we can see that it clearly speaks to the *documentary*, or *communicative* value proposition of asserts, whereby they are used to communicate important assumptions to other developers examining the same code, perhaps with a view to making modifications. Because of this, we might reasonably expect that assertions are associated with *process* aspects of asserts, specially the process aspects that relate to *collaboration*. Collaborative and human process aspects have long been a concern in empirical software engineering. Previous research [39], [16], [4] has explored the effect of factors like ownership, experience, and number of developers on software quality. Given the communicative value of asserts, it would be interesting to investigate whether collaborative aspects of software relate to assertion use. Fortunately, modern version control systems afford the reliable and straightforward measurement of process properties such as ownership and experience.

RQ2. How does assertion use relate to the collaborative aspects of software engineering, such as ownership, experience, and number of committers?

While process factors (the *how* of software) are important for assertion placement, one can certainly expect that the *product* also matters; programmers' decisions on where to place asserts will almost certainly be influenced by *what* software they are building, and *which element of it* they are working on. There are certainly a great many properties [17] of software and software elements, relating to size, complexity, coupling, cohesion, *etc*., but a comprehensive examination of all is beyond the scope of our work. In this paper, we focus on one specific aspect of assertion placement: *inter-procedural* aspects that relate to assertion placement. In particular, we focus on call-graphs.

Call graphs are a useful abstraction that capture the modular dependencies in programs. In our study, all the projects are `C` and `C++` based. Since `C++` is an object oriented language, it is not always feasible to statically determine accurate call graph of `C++` programs due to OO properties like inheritance and polymorphism. So we focus on `C` call-graphs; nodes are `C` functions, and a directed edge from function f_1 to f_2 exists if f_1 explicitly calls f_2. Static tools can build call-graphs by analyzing source code; but these are clearly approximations, when calls through function pointers are present. Nevertheless call-graphs are widely used for various empirical studies [50], [45], [20]. They have also been extensively used in *architecture recovery* [19], [12], [33], [36], [29].

We begin here with the hypothesis that the placement of assertions within functions relates to the *role the function plays in the overall system*, and that this *role is captured in the architecture of the call-graph*. While this is indeed a strong assumption, the prior use of call-graphs for architecture recovery provides some justification. The more important, or central, a method, we expect the more likely it is that developers will be inclined to place asserts therein. The field of network science [5] has produced a variety of algebraic approaches to obtain numerical measures of centrality of roles played by nodes in a network. These measures include *local* measures, such as in-degree and out-degree and *global* measures, such as betweenness centrality, Kleinberg's *hub* and *authority* measures, etc.[2] By determining the association between nodes' importance in a call graph with outcomes of interest, one can gain intuitions about which aspects of the network position of a node are most strongly associated with the given outcome. For example, nodes with high centrality in biological networks are related to organism survival [24]. Similarly, in sociology high centrality corresponds to higher social capital [47]. These types of measures have also been applied before to software dependency graphs [50]; thus it reasonable to effect that such measures may well prove strongly associated with assertion placement.

RQ3. What aspects of network position of a function in a call-graph are associated with assertion placement?

Finally, as mentioned earlier, the *application domain* of a software system may be expected to be related to assertion use. We have recently categorized GitHub projects into six general and disjoint domains, including Databases, Libraries, etc. [40] As code in these different categories may be substantially different [40], it is reasonable to expect that the code development process, including debugging, may be different across these domains. While there we did not find a relationship between code quality and application domain in our prior work, [40], assert use might be related to the domain. Thus, we ask,

RQ4. Does the domain of application of a project relate to assertion use?

III. Related Work

Assertions have a long history [10] and have been a subject of great interest, specially in the area of tools and methods to *a)* generate assertions (*e.g.,* Daikon [14]) *b)* assertion checking [41], [49] and *c)* verification [9]. Assertions have also influenced language design notably `Eiffel` [31] which introduced the notion of "design by contract". Our goal in this work is an empirical analysis of assertion use in a large program corpus; so we confine our related work discussion to work on empirical analysis of assertion use. These empirical studies fall into two broad categories: descriptive studies of the kinds of assertions in practical use, and the studies of quality impact of assertion use.

[2] We have used Kleinberg's hubs and authorities successfully earlier for finding methods relevant to a given method [42] as they are well suited to software call graphs; they provide helpful ways to identifying nodes that play an important role in the global flow of dependencies within a directed graph.

A. Studies of Assertion Usage

There have been several studies on the general usage of assertions and contracts in open source and proprietary systems. Chalin performed two studies to understand assert usage. First was a survey of 200 developers concerning assertion usage, and how errors within invariants themselves should be reported. He found that 80% of the developers used asserts in their coding at least occasionally [8]. This was then expanded upon in a study on 85 Eiffel projects, including projects from open source, Eiffel libraries, and proprietary software. He found about half the asserts were preconditions, followed by postconditions at 40%, and about 7.1% were invariants. Just over a third were null checks. As a percentage of lines of code, assertions made up 6.7% of the libraries, 5.8% of the open source projects, and 4.2% of the proprietary software.

Jones *et al.* studied 21 Java, Eiffel, and C# projects that consciously made extensive use of asserts [15]. They found that the number of asserts scaled with project size, and that assertions changed less frequently than the other code. They also reported only minor differences between the use of preconditions, postconditions, and object invariants, though preconditions tended to be more complex than postconditions. Our work is complementary; we studied usage of assertions in popular open-source `C`, `C++` projects, with no specific commitment to assertion usage, as opposed to carefully choosing projects with high usage. Our interest is to study how regular developers use assertions in a daily basis. Also, as opposed to Jones *et al.* we noticed volatile nature of assertions with significant number of deleted or modified assertion (64.59% of total added assertions are deleted or modified).

Researchers have also compared automatically generated assertions with those that developers write. Polikarpova *et al.* did a small study comparing Daikon generated assertions with developer written assertions in 25 classes. They found that while Daikon generated more valid assertions than those written by developers, it could not recreate all the assertions written by developers. Additionally, about a third of the generated assertions where not correct or not relevant [37]. Schiller *et al.* studied Microsoft Code Contracts [43] in 90 C# projects in order to understand how to help developers use them more effectively and also used Daikon to automatically generate assertions for these projects. They found most developer written assertions were NULL-check preconditions, and that Daikon generated more potential postconditions than the developers.

B. Quality Impact of Assertion Use

Do asserts and contracts help developers identify the source of a fault, once an assertion identifies an invalid system state? Early work explored using syntactic mutations to identify less testable code regions where internal invalid states might not be observable in the output and then adding asserts in such regions [46]. Several later studies [2], [6], [44] all used syntactic mutations to introduce new errors and examined how well asserts could detect and isolate the faults. Shrestha *et al.* found significant improvement in assertions detecting the mutated errors over the basic runtime error detection in Java using JML [27], a Java assertion library. The runtime error detection found only 11% of the faults, but the assertions found another 53% of the faults missed by the basic runtime checker [44]. Briand *et al.* compared the ability to identify the source of a mutated fault with and without assertions using the of number of methods between where the error was detected and the line responsible as a metric of diagnosability. They found adding assertions improved the diagnosability significantly [6]. Baudry *et al.* found significant increases with diagnosability when adding assertions as well, but also found a upper bound on improving diagnosability by just adding more asserts, suggesting that the quality of asserts was more important [2]. Our study differs in that it looks at bugs at a higher granularity and is not concerned with diagnosability. We also focus on actual bugs and not bugs induced by mutation, and our sample size is much larger than any of these studies.

Additionally, there a study comparing asserts and N-version programming abilities to detect errors [28]. They compared the additions of assertions by 24 graduate students with 2 and 3 version voting to determine the effectiveness of each in error detection. They found both method identified similar, if different numbers of faults, but that the assertions were better able to pinpointing the errors and providing useful information.

Muller *et al.* used APP [41] and jContract [23], extensions that add assertions to C and Java in two experiments with computer science graduate students to see how assertions affected the quality of output and the effort of the programmers. They looked at instances where programmers were extending existing code and writing new code. They found some evidence that the assertions decreased programmer effort in extending existing code but the reverse was true in new code. They also found that the assertions increased method reuse and that they slightly improved reliability [34]. However, the small size of the experiment limited the significance of their results and its generalizability.

Most closely related to our work was a small a case study by Kudrjavets *et al.* on two Microsoft projects comparing the density of asserts with the density of bugs in the files. They found a small negative correlation between assert density and fault density, where as the density of asserts increases the fault density decreases [26]. We extended this study and confirmed their findings.

IV. Methodology

A. Study Subjects

To understand usage patterns and code quality effects of asserts in a representative set of projects, we decided to use the 100 most popular GitHub projects, written primarily in `C`, `C++`, or both. Among these, we excluded projects where fewer than 10 asserts were ever added. This left 69 projects with 15,003 distinct authors, with 147,119 distinct files and 689,995 methods with project histories that dated back as far as 1991. Table I shows a summary of the projects we used in this paper, including ‘Linux’, ‘gcc’, ‘mongodb’ , ‘opencv’, ‘php-src’, ‘git’, ‘numpy’ etc. While assertions have appeared in about 4.6% of methods overall, the assertions appear far

more frequently in `C++` methods, with about a rate of 10.7% in comparison to a rate of only 2% in the `C` methods.

TABLE I: **Study Subjects. *Total* represents number of commits with at least one added line. *Assertion* represents total number of commits with at least one assertion in the added lines.**

		c	c++	Overall
Project Details	#Projects	63	53	69
	#Authors	13,106	3345	15,003
	KLOC	21,909	13,353	35,262
	#Files	82,462	64,657	147,119
	#Methods	472,596	217,399	689,995
	#Assert Methods	9,376	23,288	32,664
	Period	5/91 - 7/14	9/96 - 7/14	5/91 - 7/14
#All Commits	Total	4,855,798	64,657	7,035,248
	Assertion	13,751	22,374	35,901
#Bugfix Commits	Total	100,036	21,664	119,831
	Assertion	1,938	2,566	4,461

B. Data Collection

Retrieving Project Evolution History: For all projects above, we retrieved the full history all non-merge commits along with their commit logs, author information, commit dates, and associated patches. Most of the data collection was done in May, 2014. We used the command `git log -U1 -w`, where the option `-U1` downloads commit patches and `-w` outputs method names for which the code has been added. We then removed commits not affecting `C` and `C++` source and header files. Next, we marked files either *test* or *source* file, depending on the presence of the keyword 'test' within the file names. We disregard all the 'test' files from our analysis, because the use of assertions in a test context is difference than in source, the focus of this paper. We further identified bug fix commits made to individual projects by searching their commit logs for these error related keywords: 'error', 'bug', 'fix' , 'issue', 'mistake', 'incorrect', 'fault', 'defect' and 'flaw', using a heuristic similar to that developed by Mockus and Votta [32].

We implemented an assert classifier that collects assert specific statistics from commit patches by searching for the keyword "`assert`". We ignored the case of this word and included it also when it was a substring of a larger method name in order to not only capture the standard `C` assert function, but also various developer created macros and assert functions specific to individual projects. For example, the project `gcc` frequently uses functions like gcc_assert or DBUG_ASSERT as opposed to standard assert statements. Additionally, we first removed source code comments from the patches. [3] Finally, we collected the number of assertions added and deleted per commit, per project, by parsing the added and deleted lines, respectively, from each commit patch.

To evaluate the precision of the assert classifier, we selected 100 random segments of commit patches that were marked as containing asserts. No more than three commits were taken from each project to minimize project specific bias. We then manually checked the actual number of asserts added and removed in each segment. If an exact match was found, that instance was marked as correctly labeled, otherwise it was marked incorrectly labeled. The initial precision was around 90% which we improved by eliminating asserts in comments and headers. After the improvements, our final classifier had a precision of 95-98% across all projects. Of the mislabeled ones, manual examination showed that two cases out of the hundred were not asserts, and three cases were ambiguous.[4]

Collecting Process Statistics: To see the number of asserts added and deleted to a method over its lifetime, we sum the asserts added in each method on a per commit basis using the text parser described above. We similarly find the total lines and removed per method, as well as calculate the total number of commits and committers to each method. We collect these statistics both for the methods themselves and for the individual developers who contributed to each method.

Retrieving the Call Graph: To investigate where asserts are used *w.r.t.* a project's overall structure, we gathered function-level call graphs for 18 different `C` projects from their repository versions, at data gathering time. We did not attempt call-graph derivation for `C++` programs, due to complications arising from virtual-function dispatch.

First, using LLVM's *clang* tool[5], the front-end for the LLVM compiler, we parsed `C` source files to collect the names of all functions present in each. We adapted the `PrintFunctionNames` Pass that comes with the LLVM distribution to implement this step.

Second, for each of the 18 projects, we built a *Cscope*[6] database for all `C` files, containing project specific symbols and their dependencies, including function level caller-callee relationships. Such databases can be used to browse source code of very large projects like Linux, gcc etc.

Third, we combined the results from the two steps above: for each function found by *clang*, we queried the corresponding Cscope database to retrieve caller-callee information associated with it. In particular, for a queried method, option `-2` was used to find functions called by it, and option `-3` was used to find functions calling it. We merged the caller and callees for each function to build a function level call graph.

We further estimated the size in terms of SLOC of each function. This was necessary because size can be a confound in our network analysis step. For example, large functions may make many calls to other functions, and thus can have higher out-degree. Therefore, network measures such as node degree, betweenness, and hubs/authorities may correlate with function size. To address the effect of this potential confound, first we removed the commented code from the function body. Then we measured size of the functions using ctags[7] which retrieves the line number of different elements in `C` files. We extracted the line numbers of functions, structures, define statements, and typedefs for each file, sorted them based on line number, and estimated the size in LOC of each function by subtracting

[3] We disregarded the context of changes that represent unchanged source code, since we were only interested in evolutionary aspects of assertions.

[4] One of the two false positives was a comment that slipped through the filtering, and the other was a #define statement that specified assertion behavior, but was not itself an assert. The three ambiguous cases were functions related to asserts, or potential asserts, implemented in non standard ways.

[5] http://clang.llvm.org/

[6] http://cscope.sourceforge.net/

[7] http://ctags.sourceforge.net/

its starting line number from the line number of the next marked element. Obviously, this is only a rough estimate, so we randomly selected 5 or 6 functions from each of the 18 projects to obtain 100 total samples and manually checked if the approximated LOC was within a margin of error of 5 lines. In 91 cases this was true, and in none of the observed error cases was the estimate extremely different from the actual size. Therefore, this estimate is an appropriate measure for roughly distinguishing between different sized functions.

C. Statistical Methods

We use statistical tests and statistical regression modeling to test hypotheses and answer our research questions, in the R statistical environment [38]. To test for a difference in the means between two populations we use the non parametric Wilcoxon-Mann-Whitney test, for unpaired samples, and the Mann-Whitney paired test for paired samples. We interpret the results using *p*-values, indicating the likelihood of a hypothesis being true by chance, and supplement those with the Cohen's d effect size values [11]. Boxplots are used to visualize different populations.

Regression models are in general used to describe the effects of a set of predictors on a response, or outcome variable. In this paper, we use multiple linear regression and generalized linear regression [11] to model the effect of the number of asserts per method commit on outcomes, e.g., defects, related to software projects. Our data presents special challenges: most of our predictors are counts (of asserts, developers, and defects) and an overwhelming number of commits to methods has neither asserts, nor defects, i.e. the number of zero values overwhelms the non-zero values. Fitting a single multiple regression model on the entire data carries the implicit assumption that both the zero-defect/zero-assert and non-zero defect/assert data come from the same distribution, which may not be valid. Where necessary, e.g., when modeling defects as outcomes, we deal with this issue by using hurdle regression models [7], in which there are two separate models. The first models overcoming a hurdle: the effect of passing from a (defect) count 0 to a count 1; the second models the effect of going from one non-zero count to another non-zero count. Typically, the two models use nonlinear multiple regression with different linking functions. The hurdle model is usually, as in our case, a logistic regression; the count is a Poisson or negative binomial regression[8] [7].

Following the regression modeling, we use analysis of variance (ANOVA) to establish the magnitude of the significant effects in the models. We get that by observing the reduction in the residual deviance associated with the variable's effect. We log-transform dependent non-count variables as it stabilizes the variance and usually improves the model fit [11]. To check for multi-collinearity we use the variance inflation factor *(VIF)* of each dependent variable in all of the models, with a threshold of 5 [11]. We filter and remove outliers in the data where noted.

[8]Neg. binomial, compared to Poisson regression, produces narrower confidence intervals on over-dispersed data with smaller number of observations.

V. Results

We organize our result reports by the research questions discussed earlier in II. We begin with RQ1, studying the effect of assertion use on defects.

RQ1. How does assertion use relate to defect occurrence?

As reported in numerous earlier studies, any study of defect occurrence is always confounded by several factors, most critically by the size of the module under investigation [13]. Size has generally been found to be strongly associated with defect occurrence, as one would reasonably have expected; we can also reasonably expect that size will be strongly associated with assert occurrence. Another oft found confounding factor in defect modeling is the number of committers; previous research reports a "too many cooks" [22] phenomenon leading to quality issues arising from increased numbers of contributors. Thus, here, we model total defects in methods as a function of total asserts, with size and developer count as controls.

We use hurdle regression modeling which entails two separate models, hurdle and count (see Methodology), and is appropriate in our case, as adding the first assert is a "hurdle" to overcome, different than adding the second, third, etc. For the hurdle model, we use a logistic regression (generalized multiple regression with a binomial variance function) to model the binary outcome of having an assert (or not having one) in a method; total lines of code added, and number of developers are controls. Each row in this model represents a project method (or other container like structure, union, and enum). We do this on the full data set of 909,421 methods and containers left after filtering extreme points. That corresponds to asking: Is there an effect of adding an assert on the presence of a defect in the method's history? With the hurdle overcome, the second, or count, model considers only those methods whose histories include at least one defect repair, and which have at least one assert added. In it, we regresses defect counts on assert counts, controlling for lines added and the number of contributors. It corresponds to the question: Looking only at the 14,432 methods with non-zero asserts added and non-zero defects reported, what, if any, is the effect of adding an assert on the number of bugs? We use quasi-Poisson regression (to account for overdispersion in the counts) with a log linking function to model the counts. In both models we log-transformed the lines of code added variable, as it exhibited a log-linear distribution, and is not strictly a count variable.

The modeling results are presented in Table II. The left column contains the hurdle model coefficients, and the right contains the count model coefficients. We note that the effect of asserts on defects is negative in both models, in alignment with popular belief that the effect of asserts is salutary, *viz.*, towards diminishing defect occurrence. The effect size of asserts is small, as seen in the ANOVA table, but highly significant in the hurdle model. The effects of the controls are much larger, as expected.

From its model coefficients we read that by adding one developer, the odds of getting a bug (if there were none already) are $exp(0.931) = 2.53$, whereas by adding one assert the odds

of getting a bug are $exp(-0.079) = 0.924$. Thus, adding a developer increases the odds of introducing a bug (if there were none) by 153%, whereas adding an assert reduces those odds by 7.6%.

The effect of asserts on bugs in the count model is almost insignificant, and the magnitude of the effect is negligible overall. Both models together indicate that adding the first assert to a method has a significant and somewhat non-trivial effect on bugs, but after the first, on average for all developers, adding additional asserts has no appreciable difference. The variance inflation factor was well-controlled in both models.

We further examined the smaller data set of 14,432 methods with non-zero asserts and non-zero defects, to study how the presence of asserts in a method relates to defects when there are more, or fewer developers to that method. To that end, we split the data into two parts around the median number of developers per method, and then applied the count model from the right side of Table II to each. The results are in Table III. They show that, interestingly, adding asserts has a significant mitigating effect on defects when more developers are involved. In fact, even adding more lines of code seems not to be a risk for having defects in this case, suggesting a salutary effect of assert use on defects; perhaps asserts are a particularly useful aid for communication when many, perhaps less experienced "minor contributors" [4] are involved; we plan to pursue this in future research.

On the other hand, asserts seem not to matter when fewer developers commit to a method. The ANOVA analyses indicate that the size of the effect of asserts on defects is small compared to that of number of developers, but is magnified for the group with higher number of developers per method. Again, the VIF of the model variables are below 5.

TABLE II: **Bug analysis model. "T_a_LOC" stands for "Total added Lines of Code". The Count model is for the dataset: (total asserts >0 & total_bug >0)**

	Dependent variable:	
	total_bug (as binary)	total_bug
	logistic	*glm: quasipoisson link = log*
	(Hurdle Model)	(Count Model)
log(T_a_LOC)	0.052*** (0.002)	0.066*** (0.005)
developers	0.931*** (0.004)	0.212*** (0.006)
total asserts	−0.079*** (0.003)	−0.014* (0.008)
Constant	−0.972*** (0.004)	0.770*** (0.025)
Observations	909,421	14,432
Log Likelihood	−495,929.8	
Akaike Inf. Crit.	991,867.7	
Note:		*p<0.1; **p<0.05; ***p<0.01

ANOVA for Hurdle Model

	Df	Deviance	Resid. Df	Resid. Dev
NULL			909420	1119727.77
log(T_a_LOC)	1	30471.19	909419	1089256.59
dev	1	96512.46	909418	992744.12
total asserts	1	884.42	909417	991859.70

ANOVA for Count Model

	Df	Deviance	Resid. Df	Resid. Dev
NULL			14431	30725.07
log(T_a_LOC)	1	1431.49	14430	29293.58
dev	1	2780.05	14429	26513.53
total asserts	1	8.22	14428	26505.31

TABLE III: **Model explaining behavior of asserts and total bugs in methods touched by higher & lower numbers of devs.**

	Dependent variable:	
	total_bug	
	(**More** Developers)	(**Fewer** Developers)
log(T_a_LOC)	−0.024** (0.009)	0.119*** (0.007)
dev	0.211*** (0.009)	0.036 (0.035)
total asserts	−0.047*** (0.012)	0.016 (0.010)
Constant	1.257*** (0.043)	0.408*** (0.037)
Observations	5,351	9,081
Note:		*p<0.1; **p<0.05; ***p<0.01

ANOVA for **More** developer model

	Df	Deviance	Resid. Df	Resid. Dev
NULL			5350	13839.14
log(T_a_LOC)	1	31.73	5349	13807.41
dev	1	1341.43	5348	12465.98
total asserts	1	42.85	5347	12423.12

ANOVA for **Fewer** developer model

	Df	Deviance	Resid. Df	Resid. Dev
NULL			9080	14137.52
log(T_a_LOC)	1	790.68	9079	13346.84
dev	1	1.54	9078	13345.31
total asserts	1	4.93	9077	13340.37

1) Case Study: Building on the quantitative study above, we present some case studies of asserts being added in bug fixing commits in our dataset. We chose cases from two projects: Linux and Mysql. We manually examined 46 and 50 commits that had been marked as bug fix commits and have added asserts from each project respectively. We chose these two projects because both are popular and well established; both extensively use assertions, and both are from relatively different domains. By reading the bug fix commits, bug reports from Mysql bug database[9] (the examined Linux commit messages did not regularly document bug ids), and the associated patch, we manually determined whether the assert was supporting the bug fix. Out of 96 commits that we manually examined, we could relate 48 of them where asserts were added to support bug fixes. In rest of the cases, there was either a mislabeling of commits or we could not relate assertions directly to the bug; sometimes the commits were too large or addressed multiple issues which made the assert's relation to a bug fix unclear.

Table IV shows several examples of bugfix commits where assertions are used in conjunction with bug fixes to help prevent future errors. In Example 1, there were invalid memory reads caused by reading past the end of a buffer. An assert was added in the bugfix to ensure that future reads do not read from unallocated memory (see the commented lines in the Table marked in blue). The second example is of an assert checking the system state. This commit fixed a group of related bugs where the SQL command "load data infile" was causing problems—either hanging or leading to missing data if the operation was performed after a delete. Part of the fix included restarting the transaction during the end of a bulk insert, and the assert is added into to make sure that resetting successfully completes. Here, the assert does not check for the existence of the old error, but helps to protect against new errors occurring. Finally, the third example uses an improbable condition assert.

[9]http://bugs.mysql.com/

TABLE IV: **Examples of asserts added to assist with bug fixes from Mysql. The ellipsis indicates code changes omitted for space reasons. The lines started with '+' indicates added lines and started with '-' indicate deleted lines in a commit patch. A majority of asserts in Mysql use the macro DBUG_ASSERT, a part of its DBUG package. The asserted statements are marked in red.**

Example 1: Assert added to check for memory error

Author: Alexey Kopytov , Date: 2009-07-28 File: `sql/net_serv.cc`
Log Summary: Fix Bug #45031 - Allocate an extra safety byte to network buffer for when uint3korr() reads the last 3 bytes in the buffer.

```
 -921,2 +923,9  my_real_read(NET *net, ulong *complen)
       {
+          /*
+            The following uint3korr() may read 4 bytes, so make sure we don't
+            read unallocated or uninitialized memory. The right-hand expression
+            must match the size of the buffer allocated in net_realloc().
+          */
+          DBUG_ASSERT(net->where_b + NET_HEADER_SIZE + sizeof(uint32) <=
+                  net->max_packet + NET_HEADER_SIZE + COMP_HEADER_SIZE + 1);
```

Example 2: Assert added to check system state (successful restart in this case).

Author: tomas@poseidon.ndb.mysql.com, Date: 2006-02-07, File: `sql/ha_ndbcluster.cc`
Log Summary: Working on related bugs: #17154, #17158, #17081

```
 -3048,5 +3048,19  int ha_ndbcluster::end_bulk_insert()
-    if (execute_no_commit(this,trans) != 0) {
-      no_uncommitted_rows_execute_failure();
-      my_errno= error= ndb_err(trans);
+    if (m_transaction_on) {...}
+    else   {
       ...
+      int res= trans->restart();
+      DBUG_ASSERT(res == 0);
+     }
```

Example 3: Assert added to prevent impossible condition.

Author: Tatiana A. Nurnberg, Date: 2006-02-07 File: `sql/field_conv.cc`
Log Summary: Fix for Bug #48525 where `CHECK_FIELD_IGNORE` was treated as `CHECK_FIELD_ERROR_FOR_NULL`

```
 -124,9 +124,14  set_field_to_null(Field *field)
-  if (field->table->in_use->count_cuted_fields == CHECK_FIELD_WARN)
-  ...
+  switch (field->table->in_use->count_cuted_fields) {
+  case CHECK_FIELD_WARN:             ...
+  case CHECK_FIELD_IGNORE:           ...
+  case CHECK_FIELD_ERROR_FOR_NULL:   ...
+  DBUG_ASSERT(0); // impossible
```

TABLE V: **Different usage of asserts for fixing bugs.**

Usage	Mysql	Linux
Memory/Pointer	2	2
Concurrency	3	5
Comparison to 0/ NULL	12	7
Impossible Condition	3	2
Bounds and Range Checks	4	2
System State	11	12
Planned Asserts	2	0

The original bug was caused because the code was not checking for all possible values of flag `count_cuted_fields` in the code. This was fixed by enumerating all of them in a switch statement. The assert added at the end to handle a default case where the flag is some value outside the expected set. If this set were expanded or changed in the future without this region of code being updated, the assert would assist in catching the error. These examples clearly show how asserts actually help to prevent future bugs.

In order to gain a sense of what types of asserts were being added, we further classified the asserts into seven categories, as shown in Table V. These include checks on memory and pointer validity in the assert clause, checking concurrency related artifacts like semaphore, mutex, locks *etc.*, checking for null/ 0 conditions, asserting an impossible condition the system should never reach (Example 3 in Table IV), checks on array bounds/variables range validity, ensuring valid system state by checking the value of system flags (Example 2), and planned asserts, where comments showed locations where developers wanted to add more asserts in the future.

As each bug fix commit may contain several asserts, and each assert may fall into multiple categories, the categories are not disjoint. For instance, a zero comparison assert may also be checking system state. Beyond the system state checks, we found null/0 checks to be most common, which agrees with other similar studies of asserts and contracts in general [43].

RQ2. How does assertion use relate to the collaborative/human aspects of software engineering, such as ownership and experience?

Asserts are conceptually difficult, requiring a fair bit of effort and knowledge to craft, and add to the appropriate location. We

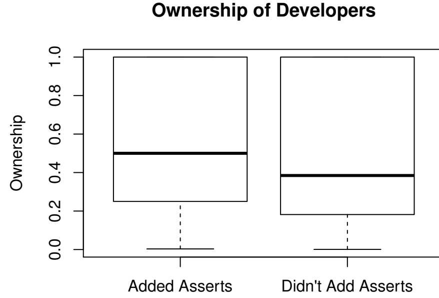

Fig. 1: **Developer ownership in methods to which they added asserts is greater. Outliers removed.**

Fig. 2: **Considering comiitters to each method, the experience of those who added asserts is greater than those who did other work. Outliers removed.**

can expect that developers adding asserts have a high degree of commitment to the specific code (types and values) as well as an algorithmic/conceptual understanding of the underlying logic. Thus we might expect that developers adding asserts to a method m have a greater degree of ownership of it than ones who just simply add code. We can also expect that those adding asserts to m have acquired some degree of skill, or experience with method m. These are related, but not identical aspects: in a very actively changed method, one might gain a lot of experience, without gaining a high degree of ownership; by the same token in a small method, one can gain high ownership without much experience. We therefore investigate the relationship of both to assertion addition separately.

We calculate *ownership* for each developer-method pair. Thus, if there are a total of 100 commits to method m, and developer d made 50 of them, then d's ownership of m is 0.5. This measure of ownership has previously been used at the level of files [4]; we extend it to the method level. We calculated ownership for all developers in a project, and all methods to which they committed. Now we separate the developers for each method m into two sub-populations; those that added asserts to m, and those that did not. We compare the ownership of each sub-population. The results are in 1. We note, first, that there is no size confound here; ownership is normalized.

Clearly assert-adding developers are associated with higher ownership; the clear visual impression is confirmed by a Wilcoxon-Mann-Whitney test (p-value $< 2.2 * 10^{-16}$). An effect-size test (Cohen's d) suggests that the effect is *small*. This supports our hypothesis that users who have greater commitment on a method will be more inclined to take the step adding asserts to it. Users not as engaged with the method will have less motivation to successfully implement an assert, and open source developers appear to follow this trend. However, one issue to note is that many methods have been changed only by a single developer, and therefore these methods have complete ownership (= 1.0). This will be more common when considering method ownership, vs, e.g., file ownership, due to the smaller granularity. Re-doing this analysis after removing methods with ownership=1.0, yields the same outcomes. That is, considering only methods with multiple developers, those who add asserts still have higher ownership.

Next we, examine the effect of *experience*. While ownership is a proportion, or fraction, between 0 and 1, experience is a cumulative measure, which generally increases monotonically with time, as a developer engages in more and more activity. We measure the experience of a developer with respect to a method m as the number of commits she has made to m. While this is, *prima facie*, a reasonable metric, it is potentially fraught with a size confound.

Clearly a larger method $\mathcal{M}$ will have more commits (and so potentially more people working on it will make commits, thus gaining more experience with $\mathcal{M}$); larger methods naturally will also tend to have more asserts. Thus a naive examination might find a spurious connection between experience and asserts, arising from the size confound. To avoid this, we compare the experience of developers on a method-by-method basis: we compare the experience of developers who add assertions, and those who don't, *for each method*.

First, we find the set of developers $D(m)$ who committed to a given method m. We partition D into $D_a(m)$, the developers who added asserts, and $D_n(m)$ the developers who did not add asserts. If either partition is empty for a given method m, it is excluded from the rest of this study. Now, for each method where both partitions are non-empty, we calculate developer experience as the number of commits made by each developer. For each method m, we then calculate the median experience of $D_a(m)$ and of $D_n(m)$.

We then get a pair of median experiences for method m, one for the developers who added asserts, and one for those who didn't. This pair can be compared without fear of a size confound, because size is implicitly controlled. The results are seen in 2. The plots shows a notable difference. A two-sample Mann-Whitney *paired* test confirms this effect (with very low p-values). A Cohen's d effect size test shows the effect to be *medium*.

RQ3. What aspects of network position of a function in a call-graph are associated with assertion placement?

This part of our work was primarily an exploratory study. Our goal was to evaluate whether the network centrality of a function had any association with assertion placement. A wide variety of ways exist to measure different properties of network positions; we tried a variety of them. In essence, we were testing a set of hypothesis as to the associations of these network centrality measures with assertion placement. For the ones that showed a significant association, we corrected the

TABLE VI: **Callgraph Centrality vs. Assertion usage**

(a) **WMW nonparametric test to compare normalized hub-score between methods with and without assertions. The p-values are from a one-sided test that methods with asserts have higher hub score than those without.**

project	functions with assert	p-value
beanstalkd	39	0.0005384
ccv	116	8.54E-63
cjdns	576	5.10E-13
firmware	66	0.00033579
gcc	5107	0
gumbo-parser	82	2.87E-09
jq	109	3.79E-11
julia	6	4.46E-05
libuv	378	2.10E-53
luvit	15	0.987256
php-src	103	0.00025826
python-for-android	85	1
twemproxy	156	1.67E-15
xbmc	558	2.59E-15

(b) **A logistic binomial regression model confirms with statistical significance that methods with assertion have more hub score, while controlling for LOC and project. Here, project is treated as dummy variable.**

	Dependent variable:	
	use_assert (as binary)	
	logistic	
LOC	0.004***	(0.0002)
hub score	7.661***	(0.151)
as.numeric(project)	−0.105***	(0.003)
Constant	−1.974***	(0.029)
Observations	83,785	
Log Likelihood	−22,529.090	
Akaike Inf. Crit.	45,066.190	

p-values to account for multiple hypothesis testing and bound the family-wise error rate.

We gathered call graphs of several project, after gathering it as described in IV-B. Note that this analysis is done on the most rescent version of the projects. We had 18 projects in total that were primarily written in `C`, for which we were able to gather call graphs (as explained earlier, `C++` call graphs are complicated by run-time despatch, which is not always statically derivable). We further removed the projects that had only 1 assert call in the entire project. This removed 4 projects, leaving us with 14 large projects that together include about 99% of all the functions from the full set of 18 projects.

To understand assertion usage *w.r.t.* project architecture, we performed the experiment in two ways. First, for each project call graph we measured in-degree, out-degree, betweenness centrality, authority, and hub scores (5 metrics in all) of each node, i.e. function. The in-degree and out-degree of a function m are counts of calls into, and calls from m. Betweenness centrality [1] of a node m is a proportionate measure of the number of geodesics passing on which m lies; it relates to the mediating role played by a function: the higher the betweenness, the more different of call-chains the function could potentially be involved in. Hub and authority are mutually re-inforcing measures of information sourcing and aggregation [25]. Hubs, essentially, represent functions which are important aggregators and dispensers of information to other functions; authorities are functions to and from which hubs despatch and collect information. Hubs and authorities are recursively defined and mutually re-inforcing: the more authorities a hub calls, the more *hub*-by it is; the more an authority is called from hubs, the more authoritative it becomes.

Several measures: out-degree, betweenness centrality, and hub-score are strongly correlated with size. Larger functions call more functions, and thus have higher out-degree; higher out-degree leads to higher betweenness centrality and hub-score. In-degree, on the other hand, is unrelated to size. We therefore normalize all the size-correlated measures by dividing them by lines of code. For each project, we then partition the functions into two groups based on whether they use an assert statement or not. Finally, we compare the normalized network metrics of the two groups using the unpaired Wilcoxon-Mann-Whitney test. In this exploration, the only measure that we found consistently related to assertions in most projects was the hub score. Table VIa shows the result of the Wilcoxon-Mann-Whitney test for the normalized hub score.

In most of the projects, functions with asserts have high hub-score with statistical significance. For only two projects (`python-for-android` and `luvit`) were the results not significant. While the former showed the opposite trend *i.e.* functions with assertion have significantly low hub score, `luvit` results remained insignificant in the opposite direction as well. Since we tested 5 hypothesis per project, very conservatively, all low p-values could be multiplied by 5 (the Bonferoni correction); all significant ones clearly remain so.

We performed similar tests for other network properties. In only five projects, functions with assertions have greater authority and in three projects they have lower authority; the rest were not statistically significant. Nothing could thus be inferred about the association of assertion usage with authoritativeness in a call graph. Similar inconclusive results were found for in-degree, out-degree and betweenness measures.

To confirm that developers use asserts primarily in the hub functions, we further performed a logistic regression on the dataset (see Table VIb). Each row in the logistic regression corresponds to a function per project. The dependent variable is `use_assert`, a binary variable, indicating whether a function is using at least one assertion. The control variables are lines of code, hub score, and project (project is treated as a categorical variable, dummy encoded). The highly significant, positive coefficient for *hub score* affirms that functions with asserts are associated with high hub scores.

To further understand why developers choose to use asserts in hub functions, we manually investigated the functionalities of several functions that use asserts and also have high hub scores. One example to describe this relationship is the `encryptHandshake` function in file `CryptoAuth.c`, which appears in the project `cjdns`, an IPv6 network encryption tool. This function is used to encrypt packets before sending them over the network, thus belonging to the core functionality of the project. This function is called by another three functions: `sendMessage`, `decryptHandshake`, and `CryptoAuth_encryptHandshake`, the first

two are important functions in the project with high authority and hub scores, respectively. For example, `sendMessage` has 5th highest authority score in `cjdns`. `encryptHandshake` in turn calls 24 other distinct functions. Thus, `encryptHandshake` turns out to be an important aggregator and dispenser of information in this context. `encryptHandshake` in turn uses asserts five times to check the validity of encrypted keys. For example, `Assert_true(!Bits_memcmp(wrapper->herIp6,-calculatedIp6, 16))` is used to make sure *they didn't memcpy in an invalid key*, as commented by the developer.

TABLE VII: **Number of assertions added to a project does not depend on the project's application domain. Domains are coded with Dummy Coding with Application domain as reference. Thus all the other domains are compared *w.r.t.* Application domain.**

		Dependent variable: total added assertion
Intercept	-0.600^{**}	(0.300)
total added lines	0.515^{***}	(0.147)
total developers	-0.250	(0.198)
project age	0.117	(0.158)
CodeAnalyzer	0.549	(0.444)
Database	0.614	(0.531)
Framework	-0.152	(0.441)
Library	0.194	(0.411)
Middleware	-0.311	(0.696)
Observations	60	
Log Likelihood	-65.072	
θ	40,563.770	(889,310.700)
Akaike Inf. Crit.	148.145	

*p<0.1; **p<0.05; ***p<0.01

RQ4. Does the domain of application of a project relate to assertion use?

In our recent work on code defects in the GitHub corpus [40], we categorized projects based on their application domain into six general groups: Applications, Code Analyzer, Database, Framework, Library, and Middleware. To investigate whether the use of asserts depended in anyway on the application domain, we used negative binomial regression due to the smaller sample size (see Methodology), with the domain as a factor, while controlling for the total number of lines, developers, and age of the project. We filtered out 9 of our 69 projects in which there were only 1 or 2 developers, or which had 30000 or more asserts added, to control for outliers, leaving us 60 observations. Out of those 17, 6, 4, 14, 13, and 6, were in the respective domains above. The 6 application domains were encoded as categorical variables, using dummy coding, with the Application domain as the reference category. The results are shown in TABLE VII. We observe that at this level of coarse modeling, only the total number of lines is a significant positive predictor for the number of asserts, as expected. While the coefficients in front of the domain variables range from negative to positive, none of them are significant. We conclude that there is no appreciable effect of the application domain on the asserts added, when controlling for code length, age, and number of developers. As with the other models, variance inflation was well-controlled.

VI. Threats to validity

Bug Identification We used commit messages, rather than bug databases, to identify bugs. While this entails some risks of false positives and/or false negatives (as also do commit links to bug databases [3]) we felt compelled to adopt this approach, since we wanted to focus on bug fix commits arising from interactive development in addition to reported bugs.

Call Graph We use Cscope to generate call graphs, which is not infallible [35]; when methods with identical names are present in different files, Cscope cannot distinguish them. We ameliorate this by looking at a method from both callee and caller perspective. However, with large samples, and reasonable network centrality measures such as hub, we expect that call graph fidelity is not a major concern. In addition, we only examined call graphs for `C`; the results may not apply for `C++`.

Assert Identification Our detection of asserts is based on a string-matching heuristic. Besides the standard `C assert()` macro, some projects have custom-defined macros which we worked to unearth and thus detect. However, our detection technique has some issues, as outlined earlier in IV. We submit that the large-sample results we have obtained are robust enough to ameliorate concerns arising from the fidelity of our assert and bug detection methods.

General Comments It's entirely possible that developers in open-source and commercial settings use asserts differently; in general, ownership can be enforced by fiat in commercial software, not so in open-source. This might lead to differences in the use and effect of asserts. Our results may not generalize beyond open-source. Our classification of domains is not objective, and others may disagree with it.

VII. Conclusions

While assertions promise great value for automated verification, in practice developers use them, often sparingly, mostly to quickly detect and report invalid system states. Even in this limited context, our data suggests that the addition of asserts is associated with reduced defects; the data also suggests that asserts help with maintaining the quality of code even as additional developers contribute code. The data also suggests that developers add asserts to methods they have prior knowledge of, and of which they have greater ownership.

Our work essentially supports the common folklore concerning asserts: they do have a salutary effect on software quality, and appear to play a positive role in collaborative software development, when many programmers are working on the same method. In future studies, we wish to see if asserts make finding and fixing real bugs, and not just bugs introduced retrospectively by researchers, easier. Additionally, we wish to classify asserts in greater depth, and ultimately to be able to identify and create useful asserts in relevant locations.

This material is based on work supported by NSF under Grants 0964703, 1247280, and 1414172.

REFERENCES

[1] M. Barthelemy. Betweenness centrality in large complex networks. *The European Physical Journal B-Condensed Matter and Complex Systems*, 38(2):163–168, 2004.

[2] B. Baudry, Y. L. Traon, and J.-M. Jézéquel. Robustness and diagnosability of oo systems designed by contracts. In *Proceedings of the 7th International Symposium on Software Metrics*, METRICS '01, pages 272–, Washington, DC, USA, 2001. IEEE Computer Society.

[3] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu. Fair and balanced?: bias in bug-fix datasets. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 121–130. ACM, 2009.

[4] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu. Doný touch my code!: examining the effects of ownership on software quality. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 4–14. ACM, 2011.

[5] K. Börner, S. Sanyal, and A. Vespignani. Network science. *Annual review of information science and technology*, 41(1):537–607, 2007.

[6] L. C. Briand, Y. Labiche, and H. Sun. Investigating the use of analysis contracts to support fault isolation in object oriented code. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '02, pages 70–80, New York, NY, USA, 2002. ACM.

[7] A. C. Cameron and P. K. Trivedi. *Regression analysis of count data*. Number 53. Cambridge university press, 2013.

[8] P. Chalin. Logical foundations of program assertions: What do practitioners want? In *Proceedings of the Third IEEE International Conference on Software Engineering and Formal Methods*, SEFM '05, pages 383–393, Washington, DC, USA, 2005. IEEE Computer Society.

[9] P. Chalin, J. R. Kiniry, G. T. Leavens, and E. Poll. Beyond assertions: Advanced specification and verification with jml and esc/java2. In *Formal methods for components and objects*, pages 342–363. Springer, 2006.

[10] L. A. Clarke and D. S. Rosenblum. A historical perspective on runtime assertion checking in software development. *SIGSOFT Softw. Eng. Notes*, 31(3):25–37, May 2006.

[11] J. Cohen. *Applied multiple regression/correlation analysis for the behavioral sciences*. Lawrence Erlbaum, 2003.

[12] K. Cremer, A. Marburger, and B. Westfechtel. Graph-based tools for re-engineering. *Journal of software maintenance and evolution: research and practice*, 14(4):257–292, 2002.

[13] K. El Emam, S. Benlarbi, N. Goel, and S. N. Rai. The confounding effect of class size on the validity of object-oriented metrics. *Software Engineering, IEEE Transactions on*, 27(7):630–650, 2001.

[14] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, Dec. 2007.

[15] H.-C. Estler, C. Furia, M. Nordio, M. Piccioni, and B. Meyer. Contracts in practice. In C. Jones, P. Pihlajasaari, and J. Sun, editors, *FM 2014: Formal Methods*, volume 8442 of *Lecture Notes in Computer Science*, pages 230–246. Springer International Publishing, 2014.

[16] J. Eyolfson, L. Tan, and P. Lam. Do time of day and developer experience affect commit bugginess? In *Proceedings of the 8th Working Conference on Mining Software Repositories*, pages 153–162. ACM, 2011.

[17] N. E. Fenton and S. L. Pfleeger. *Software metrics: a rigorous and practical approach*. PWS Publishing Co., 1998.

[18] R. W. Floyd. Assigning meanings to programs. *Mathematical aspects of computer science*, 19(19-32):1, 1967.

[19] J.-F. Girard and R. Koschke. Finding components in a hierarchy of modules: a step towards architectural understanding. In *Software Maintenance, 1997. Proceedings., International Conference on*, pages 58–65. IEEE, 1997.

[20] S. Henry and D. Kafura. The evaluation of software systems' structure using quantitative software metrics. *Software: Practice and Experience*, 14(6):561–573, 1984.

[21] T. Hoare. Assertions in modern software engineering practice. In *2013 IEEE 37th Annual Computer Software and Applications Conference*, pages 459–459. IEEE Computer Society, 2002.

[22] T. Illes-Seifert and B. Paech. Exploring the relationship of a file's history and its fault-proneness: An empirical method and its application to open source programs. *Information and Software Technology*, 52(5):539–558, 2010.

[23] JContract. Using design by contract to automate java software and component testing, 2004.

[24] H. Jeong, S. P. Mason, A.-L. Barabási, and Z. N. Oltvai. Lethality and centrality in protein networks. *Nature*, 411(6833):41–42, 2001.

[25] J. M. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM (JACM)*, 46(5):604–632, 1999.

[26] G. Kudrjavets, N. Nagappan, and T. Ball. Assessing the relationship between software assertions and faults: An empirical investigation. In *Proceedings of the 17th International Symposium on Software Reliability Engineering*, ISSRE '06, pages 204–212, Washington, DC, USA, 2006. IEEE Computer Society.

[27] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of jml: A behavioral interface specification language for java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, May 2006.

[28] N. G. Leveson, S. S. Cha, J. C. Knight, and T. J. Shimeall. The use of self checks and voting in software error detection: An empirical study. *IEEE Trans. Softw. Eng.*, 16(4):432–443, Apr. 1990.

[29] S. Mancoridis, B. S. Mitchell, C. Rorres, Y. Chen, and E. R. Gansner. Using automatic clustering to produce high-level system organizations of source code. In *International Conference on Program Comprehension*, pages 45–45. IEEE Computer Society, 1998.

[30] S. McConnell and D. Johannis. *Code complete*, volume 2. Microsoft press Redmond, 2004.

[31] B. Meyer. Applying "design by contract". *Computer*, 25(10):40–51, Oct. 1992.

[32] A. Mockus and L. G. Votta. Identifying reasons for software changes using historic databases. In *ICSM '00: Proceedings of the International Conference on Software Maintenance*, page 120. IEEE Computer Society, 2000.

[33] H. A. Müller, S. R. Tilley, and K. Wong. Understanding software systems using reverse engineering technology perspectives from the rigi project. In *Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research: software engineering-Volume 1*, pages 217–226. IBM Press, 1993.

[34] M. Muller, R. Typke, and O. Hagner. Two controlled experiments concerning the usefulness of assertions as a means for programming. In *Software Maintenance, 2002. Proceedings. International Conference on*, pages 84–92, 2002.

[35] G. C. Murphy, D. Notkin, W. G. Griswold, and E. S. Lan. An empirical study of static call graph extractors. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 7(2):158–191, 1998.

[36] G. C. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *ACM SIGSOFT Software Engineering Notes*, volume 20, pages 18–28. ACM, 1995.

[37] N. Polikarpova, I. Ciupa, and B. Meyer. A comparative study of programmer-written and automatically inferred contracts. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pages 93–104, New York, NY, USA, 2009. ACM.

[38] R Development Core Team. R: A language and environment for statistical computing, 2008. ISBN 3-900051-07-0.

[39] F. Rahman and P. Devanbu. Ownership, experience and defects: a fine-grained study of authorship. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 491–500. ACM, 2011.

[40] B. Ray, D. Posnett, V. Filkov, and P. Devanbu. A large scale study of programming languages and code quality in github. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '14. ACM, 2014.

[41] D. S. Rosenblum. A practical approach to programming with assertions. *IEEE Trans. Softw. Eng.*, 21(1):19–31, Jan. 1995.

[42] Z. M. Saul, V. Filkov, P. Devanbu, and C. Bird. Recommending random walks. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 15–24. ACM, 2007.

[43] T. W. Schiller, K. Donohue, F. Coward, and M. D. Ernst. Case studies and tools for contract specifications. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 596–607, New York, NY, USA, 2014. ACM.

[44] K. Shrestha and M. J. Rutherford. An empirical evaluation of assertions as oracles. In *Proceedings of the 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, ICST '11, pages 110–119, Washington, DC, USA, 2011. IEEE Computer Society.

[45] A. Tosun, B. Turhan, and A. Bener. Validation of network measures as indicators of defective modules in software systems. In *Proceedings of the 5th international conference on predictor models in software engineering*, page 5. ACM, 2009.

[46] J. Voas and K. Miller. Putting assertions in their place. In *Software Reliability Engineering, 1994. Proceedings., 5th International Symposium on*, pages 152–157, Nov 1994.
[47] M. M. Wasko and S. Faraj. Why should i share? examining social capital and knowledge contribution in electronic networks of practice. *MIS quarterly*, pages 35–57, 2005.
[48] S. Yau and R. Cheung. Design of self-checking software. In *ACM SIGPLAN Notices*, volume 10, pages 450–455. ACM, 1975.
[49] H. Yin and J. Bieman. Improving software testability with assertion insertion. In *Test Conference, 1994. Proceedings., International*, pages 831–839, Oct 1994.
[50] T. Zimmermann and N. Nagappan. Predicting defects using network analysis on dependency graphs. In *Proceedings of the 30th international conference on Software engineering*, pages 531–540. ACM, 2008.