Replication of *Assert Use in GitHub Projects*

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

I. Introduction

Replication is an important aspect of empirical science. In medicine, for example, there is a great deal of effort spent on replication. The *evidence-based medicine* movement gives highest weight to research that has been repeated several times, with consistent results. When multiple studies provide consistent results, they are collected into online compendia such as the Cochrane Collaboration [1] or the National Guidline Clearinghouse [2]. Replications are most warranted for surprising results [6], small samples [2], or small effect sizes [8], [1].

While replications are useful for validating results, they are unfortunately uncommon in software engineering, due to complexities associated with the software development environment [7].

In our paper *Assert Use in GitHub Projects* [4], we addressed several questions relating who added asserts to projects, where they were added, and their relationship with code defects.

In that paper, one of results (that Asserts are negatively associated with bugs) had a small effect size (< 1%), and was significant. This *per se* [1] provides a strong motivation for replication. In addition, while investigating further properties of asserts, we discovered a deficiency in the tool used to extract names of functions from changes in git logs. Since the observed effect sizes of the relationship between assertions and defects in our original study were small, we are using an exact, dependent replication [9] to determine what, if any, models change when using a more accurate method of function name extraction.

TABLE I: A sample of the git diff output.

```
@@ -104,2 +108,5 @@ static void pick_seeds( ...
+  *seed_a = node;
+  *seed_b = node + 1;
+
   for (cur1 = node; cur1 < lim1; ++cur1)
```

To demonstrate what was faulty with our original function name extraction, first consider Table I. This chunk is an example of the output from the git diff log, which makes use of the diff utils [3] package.

The tool used the optional header, e.g. the line following the @@, to identify the function name context of a diff chunk. In the case of small changes to functions, like in Table I this correctly matches the function context of the change. However, the optional header only provides an estimate of the closest function-like context of the changes. Specifically, it returns first match to a regular expression in the unmodified lines above the changes [4].

We discovered this method fails to match a change's function context in several cases. Tables II, III, and IV provide examples of when the function name will be missing or misattributed in the optional header.

When a whole file is added or deleted, like in Table II, there are no unmodified lines, and so no context is reported. These

TABLE II: A sample of the diff output for a newly added file. No function name is recorded in the header, but function(s) may appear in the file.

```
@@ -0,0 +1,135 @@
+// Copyright 2014 The Go Authors.
+// ....
...
+static void
+ffi_callback (ffi_cif* cif __attribute__((unused)),
+              void *results,
+             void **args, void *user_data)
+{
...
+}
...
```

TABLE III: A sample of the diff output for a function with a label in it. The optional header matches the label instead of the function's name.

```
@@ -87,6 +85,5 @@ retry:
                runtime_unlock(c);
-               *pfirst = nil;
-               return 0;
+               return nil;
        }
-       s = c->nonempty.next;
+       goto retry;
```

TABLE IV: A sample diff where a function is added into the code. The name in the optional header matches the function immediately above it in the file.

```
@@ -1080,2 +1060,22 @@...ccv_convnet_update_new(...

+static void _ccv_convnet_compute_softmax(
+        ccv_dense_matrix_t* a,
+        ccv_dense_matrix_t** b, int type)
+{
+ ...
+}
```

[1] http://www.cochrane.org/
[2] http://www.guideline.gov/
[3] https://www.gnu.org/software/diffutils/

[4] See sections 2.2.3.1 and 2.2.3.2 on regular expressions and c function headings in http://www.gnu.org/software/diffutils/manual/diffutils.html

were originally reported as NA in our database and ignored as the functions containing the asserts could not be determined easily in such cases. In addition, in the case of a new function, as in Table IV the regular expression will match the function immediately above it. Finally, the regular expression used by Diff Utils is not robust, and may also match labels inside functions, such as in Table III. This regular expression can also match contexts other than a function, such as a class, struct, or namespace.

Given these deficiencies, we developed a tool to more accurately parse the function names from C and C++ logs.

Not all of our research questions question are affected by this change; we list of our research questions from the original paper below:

- How does assertion use relate to defect occurrence?
- How does assertion use relate to the collaborative/human aspects of software engineering, such as ownership and experience?
- What aspects of the network position of a method in a call-graph are associated with assertion placement?
- Does the domain of application of a project relate to assertion use?

The first two RQ's depended on the function names extracted from the logs, and as such are the ones that we replicate here. RQ 1 represents the greatest cause for concern, given the small effect size. RQ 3 is not affected as the asserts and function names as the data used to answer that question was extracted from call graphs with a different tool. RQ 4 is not affected as it does not use function level information in locating the asserts, only information about which project the asserts appeared in.

Below, we summarize the results from the new data on the affected RQ's:

- We find that the number of functions containing asserts is 7.6%, higher than the 4.6% originally reported. We still find that C++ functions contain asserts more frequently than C functions, with 9.8% for C++ and only 5.9% for C, though size of the difference is not as large as originally reported..
- The original weak results for RQ 1 now have disappeared: asserts are still significant, albeit with an opposite sign from the previous result, but their effect size is tiny. Interestingly, however, in contrast to before, now we find asserts matter much more for methods developed by fewer developers (vs. by more), and are associated with a slightly higher occurrence of defects.
- We find that developers with higher ownership and experience still are more likely to add assertions, leaving RQ 2 unchanged. However, the effect size of ownership on asserts is higher while the effect size of experience is lower than in the original study.

II. Methodology and Study Subjects

A. Study Subjects

The projects selected for this replication are exactly the same as those we used in the original study. Their properties are summarized in Table V. The notable changes are in the rows #Methods and #Assert Methods. In the original paper, we found that 4.6% of methods contain asserts. In this replication study, assertions appear in about 7.6% of functions overall $(130,928/1,717,381)$. We still find that assertions appear more frequently in `C++` functions (9.8%) as compared to `C` functions (5.9%). However, this difference is not as large as reported previously (10.7% for `C++` in comparison to a rate of only 2% `C`). Additionally, the summary table in our original paper was not clear when listing the numbers of changes to functions, labeling them as commits instead. We therefore display both the number of commits containing `C` and `C++` files as well as the number changes to functions in `C` and `C++` for various types of changes. These include the changes identified as bug fixing, those that contain assertions, and those in which both apply.

TABLE V: Study Subjects. In the bottom half of the table, 'Changes' refers to the number of changes to functions in the logs, and 'Commits' represents the number of distinct commits. The number of commits/changes for bug fixes and assertions are considered only for commits to source files. Assertions are only considered if they are in the added lines of the log. Finally, commits are marked under `C` or `C++` if they modifies least one source file of that type.

		c	c++	Overall
Project Details	#Projects	63	52	69
	#Authors	12,943	3,629	15,552
	#Files	83,399	53,964	137,363
	#Methods	947,206	770,175	1,717,381
	#Assert Methods	55,626	75,302	130,928
	Period	5/91 - 7/14	9/96 - 7/14	5/91 - 7/14
#All Commits	Total	470,401	141,091	609,087
	Assertion	11,146	16,513	27,444
#Bugfix Commits	Total	127,104	47,261	172,533
	Assertion	2,700	4,859	7,472
#All Changes	Total	3,006,098	2,147,616	5,153,714
	Assertion	76,559	111,034	187,593
#Bugfix Changes	Total	621,331	650,525	1,271,856
	Assertion	15,977	25,123	41,100

For the projects summarized in Table V, we retrieved the full history for all non-merge commits along with their commit logs, author information, commit dates, and associated patches. We present results for our research questions only on commits up until July 20, 2014, the same cutoff as in our original paper. We used the command `git log -date=short -no-merges -U1000 -function-context - \\*.c \\*.cc \\*.cpp \\*.c++ \\*.cp \\*.cxx \\*.C \\*.CC \\*.CPP, \\*.C++, \\*.CP, \\*.CXX` to produce the git logs. The –function-context option displays additional context around each change so that the function name can be parsed from the chunk of diff output. This command differs from our original process in that we explicitly invoke the function-context, but also force 1000 lines of context regardless. We used this large manual context as we observed that the git diff –function-context option failed to present entire functions when they were large and contained labels. The large context allows changes to be captured in all but the largest of functions. Finally, the list of extensions

after the git log causes the command to only display diff output for files of these types, which are C and C++ source file types. We do not consider changes to files of other types of extensions, and this set is the same as used in the original paper. As in the original paper, we exclude test files from consideration, determined by the presence of the string 'test' in the file names.

We also modified the section of our tool which identifies bug fixing commits. The original version searched the commit message for keywords related to bug fixing and errors. We added several more keywords to the identifier, but also filter out some phrases that were related to error handling code and include a few project specific improvements for bug reporting styles. [5]

To evaluate whether the commits labelled as bug fixes were labelled correctly, we randomly selected 100 commits that were marked as bug fixes. We manually examined their commit messages to determine if they were in fact bug fixes. If it was not clear from the message, linked issues and the source code were used to resolve ambiguous cases. We found 94 of the 100 cases were labelled correctly (though 4 of these were style and formatting fixes). 5 cases were wrong, which included commits that changed error handling code or commits that made use of the bug related keywords such as 'fix' in a context other than bug fixing. One case was ambiguous, we could not determine if it was a bug fix or not from the message and code comments.

To extract the names of functions that changed and to find the lines containing modified asserts, we created a tool to parse the pieces of diff output in C and C++ source files produced by our log command. The tool uses regular expressions to identify functions and alternates between two phases. In the first phase the tool searches for a function name and an associated opening {, and in the second phase it searches for a corresponding closing }. Specifically, on each opening { before finding a function name, we check if the accumulated prior lines match any of a set of regular expressions for either C or C++ functions, depending on the file type..[6] These regular expressions are designed to match most C and C++ functions, including const and template functions. Note that before processing any line for assertions or function names, we remove any comments and strings from the line.

The contents of the functions are tracked by two stacks that manage the current open contexts in the old and new versions of the program, which track the number of open brackets (and what they are associated with) in each version. We also maintain a list of classes seen in the file and use this to identify constructors and destructors, which exclude the type information our match by our other regular expressions. Also, while similar to functions, our tool does not parse Macros, and we do not treat them as functions.

[5]See method if_bug ghLogDb.py on https://github.com/caseycas/gitcproc

[6]For details on the specific set of regexes used see the files CPlusPlusLanguageSwitcher.py and CLanguageSwitcher.py on https://github.com/caseycas/gitcproc

To extract assert locations, we use a very similar method to that used in our original paper, with a few modifications. We mark any line containing an assert if it contains a case insensitive match to the keyword *assert*. Additionally, while manually checking our projects we observed that there were a handful of macros that when expanded were clearly asserts, but did not contain the keyword assert. These were the functions *ut_a* and *ut_ad*. If we saw a function call that exactly matched any of these names, we also marked it as an assert.

We then filtered out any asserts on unmodified lines, and also ignored any functions parsed from the chunks that contained no modified lines. Asserts found outside functions were grouped into one mock 'function'. These asserts were excluded from the results for RQ1 and RQ2.

We perform additional filtering on these returned names to return a set of strictly function changes. This was performed via an sql command to remove classes, labels, function definitions, and other structures. Our original paper filtered some of these, such as blank rows marked as NAs, but the current filter produces a more reliable set of function names.

Our tool was able to parse all but one of the logs used in the earlier paper, the log for php-src. The tool continuously hung on this project for reasons we could not determine. Therefore, we used the results from a successful parse of php-src from a earlier version of this tool. We had found similar precision rates for that version, but the false negative rates were much higher.

To evaluate the precision our parser of assert and function names and subsequent filtering, we selected 100 random instances of function changes that our tool had extracted and marked as having asserts added or deleted. To classify an instance as correct, it must satisfy all the following criteria. One, the function name must be correctly parsed from the log. Two, the number of assertions added and deleted must exactly match what is seen in the log file. Three, the number of lines added and deleted in the function must exactly match what is seen in the log file. In our random sample 95 examples were correct. Two of the incorrect examples resulted from complex changes to brackets that grouped a couple functions under one name. Two others correctly identified the function, but miscounted the assertions, as they did not capture the *DCHECK* function in the *v8* project, which acts as an assertion even though it doesn't share the name. The last case was a change to an function implementing assertion logic, but was did not contain a call to an assert itself.

We also evaluated 100 random samples of sections of code marked as non-functions to estimate how many functions we were not parsing. Since there may be many non functional changes in a chunk of code (@@ segment), all of which our tool will group together, we limited our sample to non function changes with less than 50 additions and 50 deletions so that they could be manually examined. We mark an example as false if there are changes in a function that were grouped into the count of the non function changes. Here, we had 80 correct examples, and 20 mistakes. 10 of these come from php-src, which was parsed with the older version of the tool.

Most of the mistakes (11, exactly), resulted from insufficient context in the git logs from very large functions with many labels. These missed functions were the result of logs similar to Table III, which had no function for our parser to extract. The others were the result of unusual function names [7] that fell outside of what our regular expressions captured. Given the total numbers of changes we marked as functions and non-functions (5,746,094 and 1,408,065 respectively if changes after July 2014 are included.), and using our sample false positive and false negative rates as estimates of the true false positive and negative rates, we estimate the tool has a approximately 95% precision and 95% recall. Finally, there are a few cases where our tool is unable to parse `C` or `C++` diff chunks. However, we observed only a total of only 10 cases, which is insignificant relative to the total number of changes.

Beyond these changes, the code and methodology used for to produce results for RQ 1 and RQ 2 remains unchanged.

The tool used to parse the log files is available at https://github.com/caseycas/gitcproc/. This tool is undergoing active development, so to see the version used for this paper, check out before the commit *6d80dd476fcc72c457de1d83926e8e0357d2f848* on the tool-eval branch. We have also made the other data and scripts used in this replication and the original paper at https://github.com/caseycas/assert_replication. This repository contains instructions that will allow you to replicate the results of this paper.

III. Results

A. RQ1. How does assertion use relate to defect occurrence?

We fit here the same hurdle model [3] as in our original paper, and investigate how our improved function extraction affects the relationship between assertions and defects. As a reminder, the hurdle model is useful in modeling count data, particularly when the response variable has a large number of zero values. A single regression model makes the assumption that both the zero-defect/zero-assert and non-zero defect/assert data follow the same distribution. To avoid this assumption, the hurdle model allows us to model the effect of going from a defect count of 0 to 1 separately from further increases in the defect count. For additional details, refer to our original paper [4].

Table VI shows these hurdle and count models with our new data. Code size and number of developers remain positively correlated with defects, with large effect sizes. However, the effect of asserts has further diminished, reversed direction, and is positively correlated with defects. The effect of asserts is very small because its coefficients are small in both models (e.g. the odds for asserts in the logistic regression is 1.03), but also because the portion of NULL deviance explained by this variable is much less than 1%.[8] We conclude from these models that there is no evidence that non test asserts have an effect on defects in general.

However, our original paper also explored the effect of assertions in functions that had many developers versus those that had only a few developers. The theory was that the invariants displayed in assertions could act as a communication between developers, helping them to avoid bugs. Table VII displays the relationship between defects and asserts for in functions with greater and fewer numbers of developers. In the model with greater numbers of developers. the relationships between asserts and defects has reversed and is significant, but again, the effect size is incredibly small. In the model with fewer developers, which was statistically insignificant in our original paper, the effect of asserts is significant and positively correlated with bugs. Moreover, the asserts explain 2.25% of the NULL deviance, showing a small positive effect between assertions and bugs in methods having few developers.

TABLE VI: Bug analysis model. Lines added and asserts added measure the number of '+' in the log diff file. The Count model is for the dataset: (total asserts > 0 & total_bug > 0)

	Dependent variable:	
	total_bug >0	total_bug
	logistic	*glm: quasipoisson link = log*
	(Hurdle Model)	(Count Model)
log(lines added)	0.178*** (0.002)	0.183*** (0.003)
dev	1.144*** (0.003)	0.214*** (0.002)
asserts added	0.032*** (0.002)	0.048*** (0.003)
Constant	−1.042*** (0.005)	−0.004 (0.012)
Observations	1,689,229	52,193
Log Likelihood	−945,740.500	
Akaike Inf. Crit.	1,891,489.000	

Note: *p<0.1; **p<0.05; ***p<0.01

	Df	Deviance	Resid. Df	Resid. Dev
NULL			1689228	2206785.97
log(lines added)	1	94459.98	1689227	2112326.00
dev	1	220524.09	1689226	1891801.91
asserts added	1	320.83	1689225	1891481.07
	Df	Deviance	Resid. Df	Resid. Dev
NULL			52192	70574.98
log(lines added)	1	14747.39	52191	55827.59
dev	1	10658.85	52190	45168.74
asserts added	1	299.59	52189	44869.15

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

Here we investigate the changes to the observed characteristics of those who add asserts to functions and those who don't. Again, we focus on the developer's ownership of and experience in the function, as defined and calculated in exactly the same manner as our original paper [4].

Figure 1 shows the relationship between a developer's ownership of a function and whether or not they added any

[7]For example, the version in this paper did not handle struct constructors, constructors and destructors without a class definition in the same chunk, overridden virtual functions, and K&R functions: http://stackoverflow.com/questions/3092006/function-declaration-kr-vs-ansi

[8]The portion of the NULL variance that is explained by an independent variable, as calculated, e.g., by an ANOVA, can be used as the independent variable's effect size [5]. As we use non-linear models, the deviance plays the role of the variance.

TABLE VII: MODEL EXPLAINING BEHAVIOR OF ASSERTS AND TOTAL BUGS IN METHODS TOUCHED BY GREATER & FEWER NUMBERS OF DEVELOPERS. ASSERTS AND LINES ADDED AGAIN MEASURE THE NUMBER OF ASSERTS AND LINES THAT ARE MARKED AS '+' IN THE DIFF OUTPUT.

	Dependent variable:	
	total_bug	
	(Greater)	(Fewer)
log(lines add)	0.187^{***} (0.005)	0.150^{***} (0.004)
dev	0.167^{***} (0.003)	-0.103^{***} (0.018)
asserts added	0.018^{***} (0.004)	0.098^{***} (0.004)
Constant	0.168^{***} (0.022)	-0.154^{***} (0.017)
Observations	18,628	33,565
Note:	*p<0.1; **p<0.05; ***p<0.01	

	Df	Deviance	Resid. Df	Resid. Dev
NULL			18627	35532.30
log(lines added)	1	6695.44	18626	28836.86
dev	1	3831.85	18625	25005.01
asserts added	1	27.30	18624	24977.71

	Df	Deviance	Resid. Df	Resid. Dev
NULL			33564	20416.64
log(lines added)	1	2402.05	33563	18014.59
dev	1	30.40	33562	17984.19
asserts added	1	459.93	33561	17524.26

asserts. As we found before, developers with higher ownership are still more likely to commit asserts to that function. We confirm the finding suggested by the boxplot using a one-sided Mann Whitney Wilcoxon test with an alternative hypothesis that the ownership of developers committing asserts is higher. This test returns a p-value of less than $2.2 * 10^16$, and a Cohen's D effect size suggests the effect is small to medium (0.349). This effect size is larger than the *small* effect size (.217) observed in the original paper.

Regarding experience, Figure 2 displays the relationship between a developer's experience in a function and whether or not they added asserts. This is calculated only for functions where asserts were added and where both developers who added asserts and developers that did not add asserts contributed code. As with ownership, our new results confirm our findings from the original paper; developers with more experience are more likely to add assertions than those with less experience. Using a *paired* one-sided Mann Whitney Wilcoxon test, this time with an alternative hypothesis that the experience of developers who add asserts is higher, we reject the null with a p-value less than $2.2 * 10^16$. The effect size is slightly smaller than for ownership, but is still between small and medium (0.354). This effect size is smaller than the *medium* effect size (0.512) obtained in the original paper.

IV. CONCLUSIONS

Empirical studies on large data sets of software projects provide the opportunity to discover small and nuanced effects which are not easy to identify in more controlled experiments. However, small effects must be interpreted carefully, particularly when using tools provide only approximations of the effects being measured. In the case of our study of asserts, we see that with the improvements to the identification of changes in functions and bug fixing commits, the very small mitigating effect of asserts on bugs reversed direction, and while still statistically significant, has an effect size too small

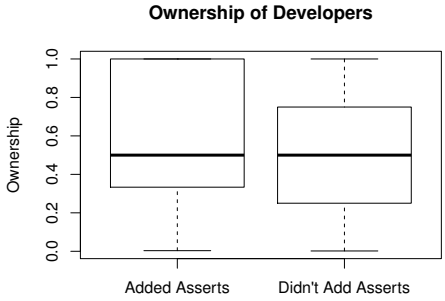

Fig. 1: Developer ownership in functionss to which they added asserts is greater. Outliers removed.

Fig. 2: Considering comitters to each function, the median experience of those who added asserts is greater than those who did other work. Outliers removed.

to be meaningful. However, results where we saw larger effects, such as ownership and experience, remain roughly same.

We hope this replication will help emphasize the importance of considering both statistical significance and effect size when reporting results. In large data sets, when the effect size is small, even if it is significant, the results should be interpreted with caution.

REFERENCES

[1] A. Arcuri and L. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 1–10. IEEE, 2011.

[2] K. S. Button, J. P. Ioannidis, C. Mokrysz, B. A. Nosek, J. Flint, E. S. Robinson, and M. R. Munafò. Power failure: why small sample size undermines the reliability of neuroscience. *Nature Reviews Neuroscience*, 14(5):365–376, 2013.

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

[4] C. Casalnuovo, A. Oliveira, V. Filkov, P. Devanbu, and B. Ray. Assert use in github projects. In *Proceedings of the 37rd International Conference on Software Engineering*, ICSE '15. ACM, 2015.

[5] J. Cohen. *Statistical power analysis for the behavioral sciences (2nd edition)*. Lawrence Erlbaum Associates., 1988.

[6] J. P. Ioannidis. Why most published research findings are false. *PLoS Med*, 2(8):e124, 2005.

[7] N. Juristo and S. Vegas. Using differences among replications of software engineering experiments to gain knowledge. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, pages 356–366. IEEE Computer Society, 2009.

[8] V. B. Kampenes, T. Dybå, J. E. Hannay, and D. I. Sjøberg. A systematic review of effect size in software engineering experiments. *Information and Software Technology*, 49(11):1073–1086, 2007.

[9] F. J. Shull, J. C. Carver, S. Vegas, and N. Juristo. The role of replications in empirical software engineering. *Empirical Software Engineering*, 13(2):211–218, 2008.