Assessing the Relationship between Software Assertions and Code Quality: An Empirical Investigation

Gunnar Kudrjavets [1], Nachiappan Nagappan [2], Thomas Ball [2]
[1] Microsoft Corporation, Redmond, WA 98052
[2]Microsoft Research, Redmond, WA 98052
{gunnarku, nachin, tball} @microsoft.com

Abstract

The use of assertions in software development is thought to help produce quality software. Unfortunately, there is scant empirical evidence in commercial software systems for this argument to date. This paper presents an empirical case study of two commercial software components at Microsoft Corporation. The developers of these components systematically employed assertions, which allowed us to investigate the relationship between software assertions and code quality. We also compare the efficacy of assertions against that of popular bug finding techniques like source code static analysis tools. We observe from our case study that with an increase in the assertion density in a file there is a statistically significant decrease in fault density. Further, the usage of software assertions in these components found a large percentage of the faults in the bug database.

Keywords: Assertions, Faults, Bug database, Source control systems, Correlations.

1. Introduction

There is much literature that makes a case for the use of assertions and discusses the potential benefits of using assertions in software development. But to date, there have been limited studies in academia or in industry that empirically address the utility of assertions. Even when we talk to developers within Microsoft there are no unified opinions about the usefulness of assertions.

Some developers use assertions in a very disciplined way and some developers don't use assertions at all.

One reason why assertions have not been investigated is due to the lack of access to industrial programs and bug databases. Results obtained from benchmark programs suffer from the small size of these programs. In industry, most of the large applications still have a sizeable amount of legacy code where the use of assertions is minimal. This leads to the lack of conclusive results in empirical analysis as it is very rare that software developers will go back into code and add assertions. This causes the analysis to be skewed as size plays a huge factor in these results.

Assertions have been studied for many years [5, 11, 13]. In 1998 there was an investigation [14] performed for the NIST (National Institute for Standards and Technology) in order to evaluate software assertions. This report raised the following generalized observation regarding assertions:

> ... *"Interest in using assertions continues to grow, but many <u>practitioners are not sure how to begin using assertions or how to convince their management that assertions should become a standard part of their testing process.</u>"*

In this paper, we focus on addressing the above underlined statement. There have been several research papers on the formal analysis of assertions in code, their utility in contracts, investigations on the placement of assertions etc. but none on the practical implications of using assertions in commercial code bases. This paper attempts to address this question from an empirical perspective by investigating the relationship between assertions and code quality for commercial software components at Microsoft Corporation.

We are able to investigate the relationship between software assertions and code quality due to the systematic use of assertions in the Microsoft components and accurate synchronization between the bug tracking system and source code versions. This makes it easy to find for each fault the number of lines and source code files that are affected. Every fault is associated with the list of code changes (if any) which are made to fix it. Based on our investigation we find that there is a statistically significant negative correlation between the assertion density and fault density. Extracting the number of faults related to assertions from the bug database indicates a substantially large number of faults. This indicates the efficacy of using software assertions.

This paper is organized as follows. Section 2 provides the related work on software assertions. Section 3 introduces the details of the case study setup and data collection. Section 4 discusses the results of the case study. Section 5 presents the threats to validity and Section 6 provides a discussion of the lessons learned and future work.

2. Related Work

In this section we discuss the related work on the use of software assertions in programming. Assertions have been investigated as a fault detection practice in software systems [8]. Rosenblum [12] defines assertions as being *formal constraints on software system behavior that are commonly written as annotations of a source text. The primary goal in writing assertions is to specify what a system is supposed to do rather than how it is to do it.* Rosenblum's study [12] presents a classification of assertions for detecting faults. The assertions are primarily of two types of specification: specification of **function interfaces** (e.g. consistency between arguments, dependency of return value on arguments, effect of global state etc.) and specification of **function bodies** (e.g. consistency of default branch in switch statement, consistency between related data etc.). Rosenblum also shows the need for a high-level automatically checkable specification of the required behavior.

Empirical research into contracts (assertions) [1] focuses on diagnosability and robustness. Diagnosability is the degree to which the software allows easy and precise location of a fault when detected. Robustness is defined as the degree to which software can recover for internal faults that would have lead to failures [1]. Baudry et al. [1] empirically validate such robustness and diagnosability factors using different measures by applying mutation analysis in a telecommunications switching system. They observe that Design by contract is an efficient way of improving the diagnosability and robustness of a system and its general quality. Similarly, Briand et al. [2] investigate in detail the impact of contract executable assertions on diagnosability. They compare the results of programs instrumented with contracts and programs that exclusively use test oracles. The ATM case study they used is 2.2 KLOC in size. Based on the mutant versions generated (by seeding faults) they compute diagnosability for program versions using only test oracles and by using instrumented versions using contracts. The average diagnosability in the ATM program by using contract is at least eight times better than using just test oracles and this implication leads to significant effort savings.

Voas and Miller [15] discuss an interesting approach to placing assertions in code. An excessive number of assertions slows down the execution speed and there might not be any cost-benefits between the performance degradation and the potential benefits of finding faults. Their work aims to place assertions in locations where traditional testing is unlikely to uncover any faults via sensitivity analysis. Sensitivity analysis makes predictions concerning future program behavior by estimating the effect that (1) an input distribution, (2) syntactic mutants, and (3) changed values in data states have on current program behavior [15].

Muller et al. [9] conducted two controlled experiments about assertions as a means for programming, using graduate students in a university. They investigate the relationship between assertions and maintenance, reuse and reliability. They observed that the programs produced by the group that used assertions were more reliable (though not statistically significant so) compared to the programs developed by the group of students not using assertions.

Our contributions are closely related to the previous studies. Compared to previous empirical analysis of assertions [1] [2] [15] we take a black box approach to investigating the relationship between software assertions and quality. Instead of mutant versions [2] to investigate the efficacy of assertions we use actual faults extracted from a bug database due to deployment of the software. Our study provides empirical evidence of the relationship between assertions and code quality using commercial software systems. Mining the bug database also allows us to determine the number of faults that were related to the usage of assertions.

3. Case Study

In our case study we investigate the relationship between assertion density and code quality (measured by post-release fault density). In Section 3.1 we describe the case study setup and in Section 3.2 the data collection process for mining the software assertions and fault information.

3.1. Case Study Setup

Our empirical case study was performed on two commercial components at Microsoft Corporation. These two components are software development tools which are part of the Visual Studio^{TM} system at Microsoft Corporation. The reasons for selection of the two components are,

- The code base was primarily written using C and C++. The whole team wrote and modified unit tests (which are run before any changes to the code) during code development. On a few occasions the team specifically went back and added additional unit test coverage where new problems were discovered. In addition the component's test team ran their automated tests nightly and obtained very high block code coverage.

- It is a sufficiently mature code base. There wasn't a formal stance on assertions during the development of components A/B. Developers in these teams recognized the value of assertions. This ensures that our results are not skewed due to an enforcement policy mandating the use of assertions.
- The components were tested with similar rigor for both the releases as these systems were released to thousands of software developers and partners within and outside the company.
- There was very accurate synchronization between the bug tracking system and source code versions. This makes it very easy to find for each fault the number of lines that are changed and source code files affected. Every fault is associated with the list of code changes which are made to fix it.

For each component, we analyzed two internal releases, thus providing four data sets to identify the relationship between assertion density and fault density. Let A-1, A-2 and B-1, B-2 denote the two internal releases of components A and B. Figure 1 shows for analysis timeline for component A-1 and A-2 .

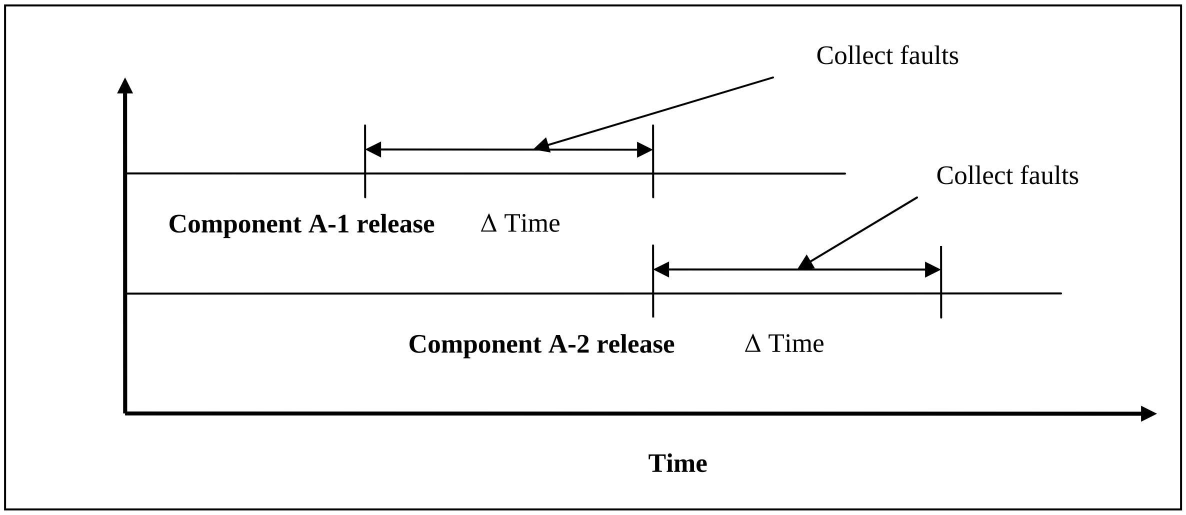

Figure 1: Analysis timeline for software components

Similarly we investigate a similar analysis timeline for component B-1 and B-2. For each component we measure the number of assertions at the point of release. After each component is released we measured the faults for the component post-release until the next version (after Δ time) is released. Using the assertion and fault information with the size of the system (in 1000's of lines of code (KLOC)) we compute the assertion density (number of assertions/KLOC) and fault density (number of faults/KLOC). More formally our research hypothesis is:

Hypothesis*: With increase in the assertion density of the code there is a decrease in the fault density.*

3.2 Data Collection

In this section we describe the data collection process for mining the assertion and fault information from the source control system and bug database to aid other researchers in replicating our studies. Figure 2 explains the data extraction methodology used in our case study to mine the faults from the bug database. We define the extracted faults according to the IEEE [6] definition as, *when software is being developed, a person makes an error that results in a physical fault (or defect) in a software element. When this element is executed, traversal of the fault/defect may put the element (or system) into an erroneous state. When this erroneous state results in an externally visible anomaly, we say that a failure has occurred*. This definition of fault is confirmed by verifying if the entry in the bug database has a corresponding source code changes as shown in Figure 2.

We follow an interactive procedure for mining the faults and assertions (Figure 2 describes this process in detail). We extract all the bug information pertaining to a component (post-release). For each entry in the bug database we check if there are any associated source code changes. If not we ignore the bug otherwise we classify the bug as a fault and extract the list of files associated with the change from the source control system change logs. From this information we check if there were any executable lines changes, if not we ignore the bug (as sometime only comments might have changed). If there are executable source code changes we extract the lines of code in the file(s) and the number of assertions. We iterate this process until all the post-release bug information associated with a component is processed. We then merge the number of assertions, faults and size information to compute the assertion density and fault density on a per file basis.

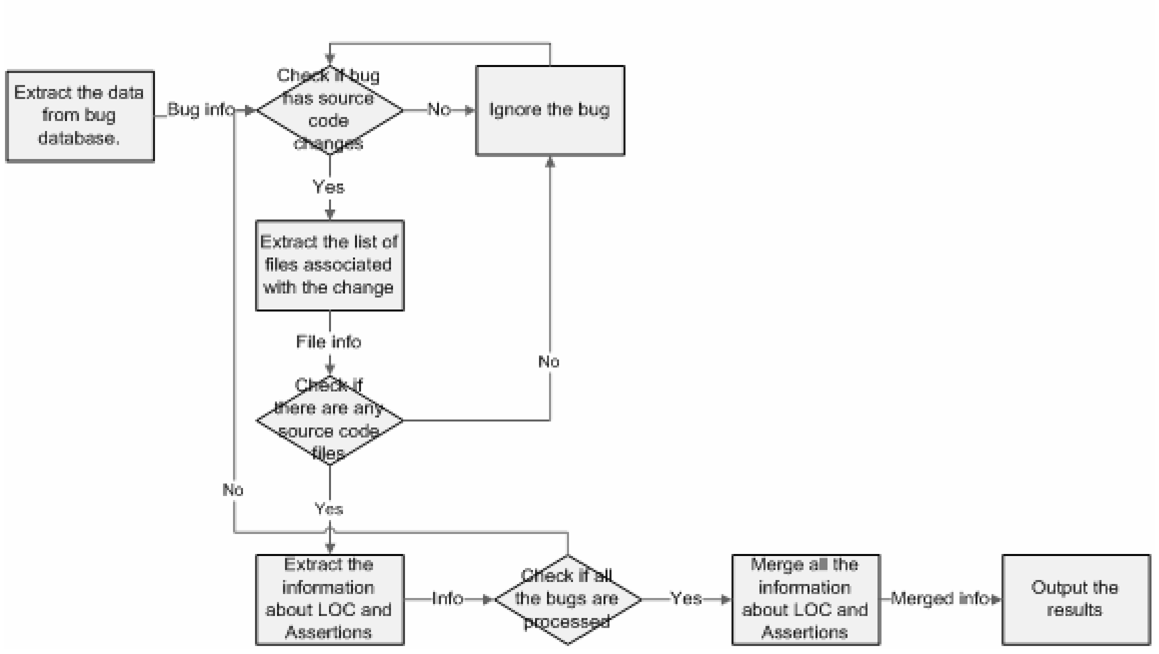
Extract the data from bug database.
Bug info
Check if bug has source code changes
No
Ignore the bug
Yes
Extract the list of files associated with the change
File info
No
Check if there are any source code files
No
Yes
Extract the information about LOC and Assertions
Info
Check if all the bugs are processed
Yes
Merge all the information about LOC and Assertions
Merged info
Output the results

Figure 2: Data extraction methodology

The extraction of the assertions and faults described in Figure 2 was completely automated via a parser written in C#. An example of the usage of invariants, pre-conditions, and post-conditions is also shown in Appendix A.

4. Case Study Results

To evaluate the relationship between assertion density and fault density we extract the software assertions and fault information for all the four releases (A-1, A-2, B-1, B-2). We evaluate the relationship between the fault density and the assertion density using a Spearman rank correlation. The Spearman rank correlation is a commonly-used robust correlation technique [4] because it can be applied even when the association between elements is non-linear; the Pearson bivariate correlation requires the data to be distributed normally and the association between elements to be linear. Since we use a rank correlation technique we remove from the analysis all the files that have no assertions and no faults. Such files can skew the results because they cannot be used to evaluate the efficacy of assertions and inflate the size of the study. We do include files that have faults but no assertions as well as files with assertions but no faults. We remove only the files that don't have both faults and assertions. Table 1 describes the sizes of components A and B. The total team size in terms of personnel for components A and B was on the order of 14-18 software developers. The average assertion density values for the four analyzed components are also shown in Table 1.

Table 1: Components size

Component	Size in terms of lines of code	Number of source files	Assertion density
A-1	104.03 KLOC	140	26.63 assertions/KLOC
A-2	105.63 KLOC	124	33.09 assertions/KLOC
B-1	372.04 KLOC	931	37.09 assertions/KLOC
B-2	365.43 KLOC	853	39.49 assertions/KLOC

Figure 3 presents the scatter plots of the assertion densities and fault densities for the files that comprise of the components A and B. The axes for the fault densities have been removed to protect proprietary information. In Figure 3 we indicate a point of interest in components A-1 and B-1 by “1” and in components A-2 and B-2 by “2”. Each dot in the scatter plot represents a unique file.

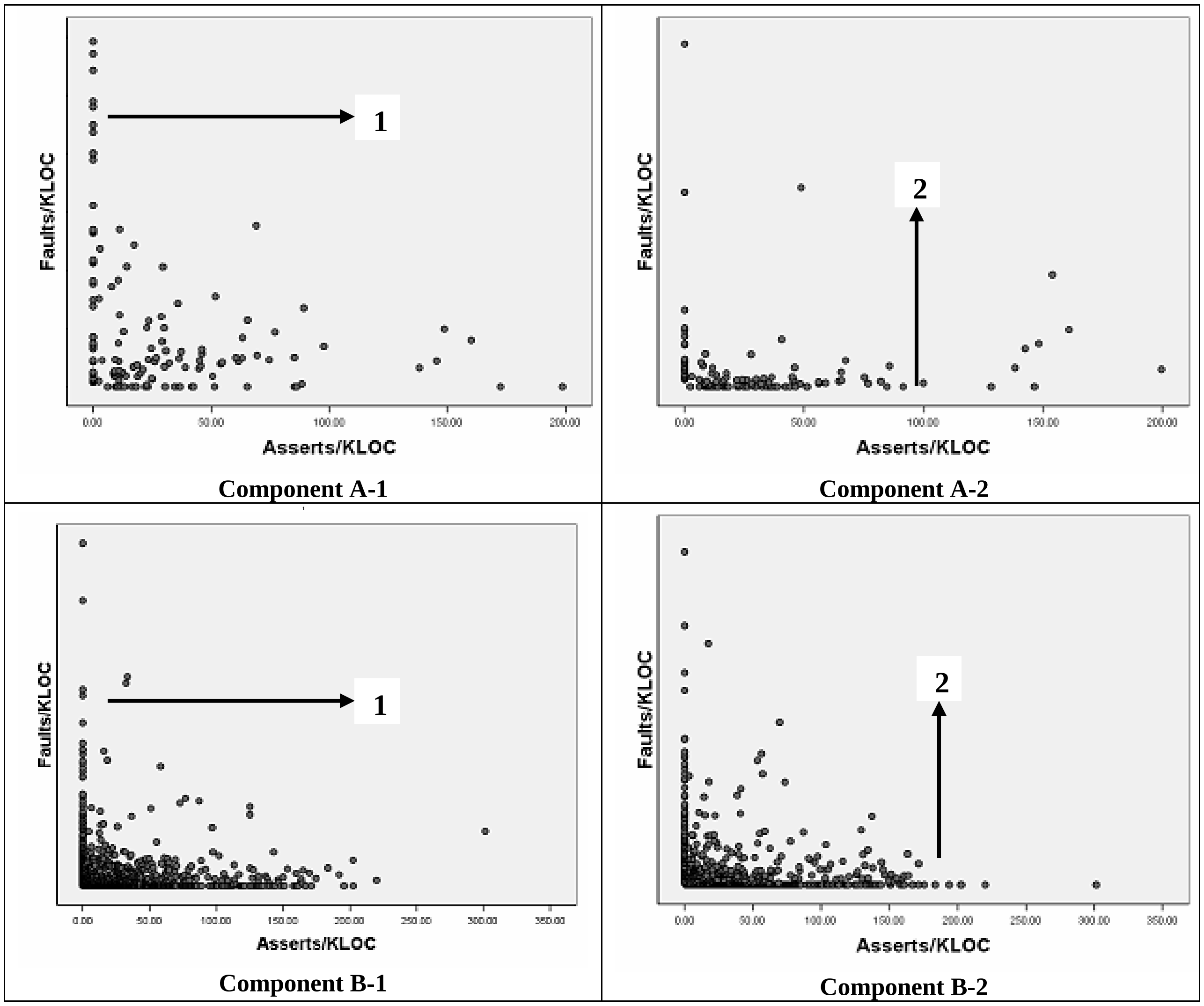

Figure 3: Scatter plots between assertion density and fault density for components A, B

Point “1” indicates the vertical trend for files that have zero assertion density indicating that there is a high fault density. Similarly point “2” indicates the horizontal trend that shows that the files with a low fault density have a higher assertion density. In order to quantify the relationships shown in Figure 3 we analyze the data using Spearman rank correlation. Table 2 shows the correlation results for components A and B. From the results of the correlation analysis we observe a statistically significant negative relationship between the assertion density and the fault density. This indicates that with an increase in the assertion density there is a decrease in the fault density and vice versa.

Table 2: Spearman rank correlation coefficients

Component	Spearman correlation (significance)
A-1	- 0.320 (p<0.0005)
A-2	- 0.209 (p<0.0020)
B-1	- 0.384 (p<0.0005)
B-2	- 0.295 (p<0.0005)

We then mined all the faults from the bug database to determine how many of these were caused by assertions (e.g. assertion failures). The utility of assertions can be evaluated by their ability to detect faults [1, 2, 15]. Upon analyzing the faults we observe than on an average 14.5% of the faults for component A and 6.5% of the faults for component B are related to assertions (Figure 4). These percentages are substantially large considering the size and complexity of the components. We then calculated the percentage of faults found by source code static analysis tools. Static analysis tools are cheap to run and have low overhead. They identify errors such as buffer overflows, null pointer dereference, the use of uninitialized variables etc. Using static analysis tools within Microsoft (FxCop, PREfast, and PREfix) [3, 7, 10] we obtain the faults from the bug database for components A and B to present a relative view of the percentage of faults found due to assertions. Figure 4 describes the relative view of the percentage of faults found by the static analysis tools and by assertions. From Figure 4 we can demonstrate the significance of the large percentage of faults found using assertions. We are not comparing the utility of static analysis tools and using assertions but just want to present a relative view to understand the scale and complexity of the systems. Based on the large percentage of faults found using assertions we see that there is a potentially huge benefit in using assertions for software development.

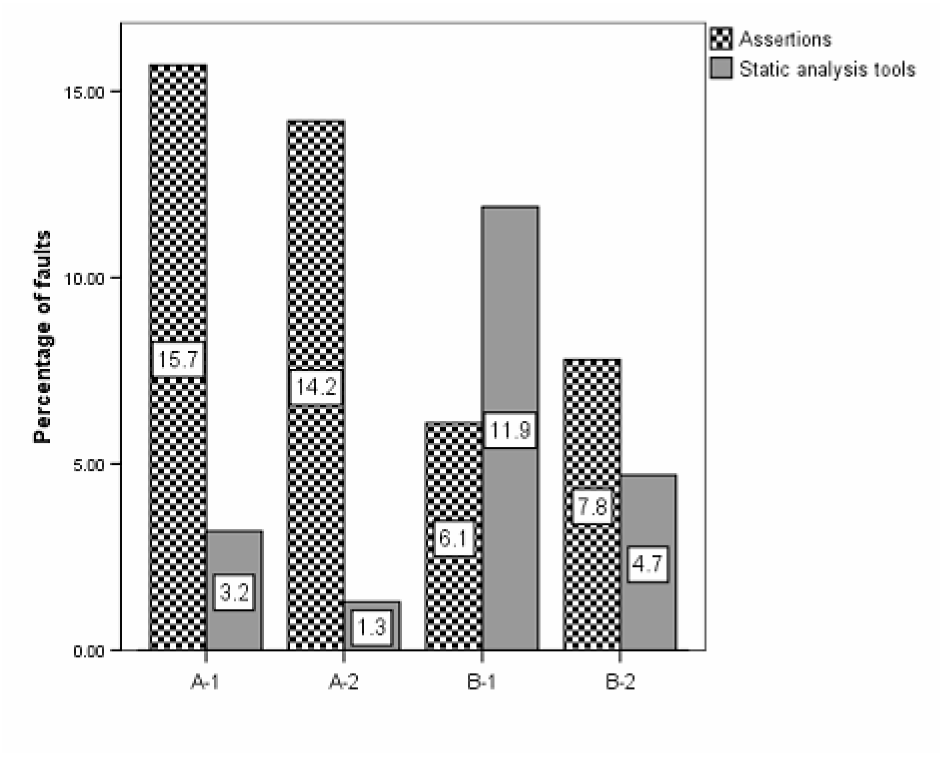

Figure 4: Fault percentages via assertions and static analysis tools

5. Threats to Validity

In controlled studies say using students one can use the results of standardized tests, GPA etc. to quantify the "effectiveness" of the subjects. Unlike controlled studies it is not possible to measure the "effectiveness" of professional developers. It could be possible that the developers in our study were better programmers than the average programmer in the industry. Unfortunately this is a threat we cannot address. But we can say that based on policies for recruitment at Microsoft to a large degree most programmers will have a similar skill levels.

We measure the assertions at one point in time, the released version of the component. It is possible that developers might have removed assertions during the development time frame which could have also improved the quality of the components but cannot be measured as we take only one snapshot of the component. Though it is highly unlikely that developers will go back to code to remove working assertions we plan on addressing this in future studies by using a moving average of the assertion numbers.

6. Lessons Learned and Future Work

Based on our empirical analysis across two large components at Microsoft we observe a similar statistically significant negative trend between the fault density and the assertion density. This indicates the uniformity of results across the components A and B. Statistically we can say that with higher assertion density there is a corresponding lower fault density. But the levels of statistical significance are not high (around 0.3's) but are in the appropriate negative direction indicating the desired trend.

This brings us to an important issue: what are the implications of these results and what action can we take based on them? We believe enforcing the use of assertions would not work well. For example, mandating that developers should have an assertion density of 30 asserts/KLOC may not lead to an effective use of assertions. We can build cross-check measures like code coverage to tie back to the assertions, i.e. when there is an increase in the assertion density is there an increase in the code coverage numbers to make sure that the new assertions are exercising new code etc.

Based on talking to different developers and managers across Microsoft our observation is that unless there is a culture of using assertions, such mandates will not produce the desired results. If the developers are mature enough to understand the code base and write useful assertions, it is highly likely that they understand the code base, which will lead to a lower fault density. We feel there is an urgent need in educating students about the utility of software assertions. We plan to collaborate with academics to help drive some of these results in the classroom.

Ideally our future work would be a planned controlled case study that investigates two teams (> 10 people in size) under the same organization. One team would use assertions in the code and the other team would not. We intend to compare their fault densities, post-release to assess the stability of the code base and indirectly their quality. The second and more important aspect is what is the cost benefit trade-off? If using assertions is going to be cost-ineffective then it would be very hard to get teams to adopt this kind of results. At a very high level we should consider this a first study to address some of the concerns of the NIST report [14]. Towards this end we intend that our case study contributes towards strengthening the existing empirical body of knowledge in this field [1, 2, 12, 14, 15]. Another point would be to repeat this study at a finer lever of granularity (say at the function level) to compare the strength of the correlation results to identify if there exists a stronger relationship between the assertion density and code quality.

Acknowledgements

We would like to thank Tony Hoare, and Jim Larus for discussions regarding the use of assertions at Microsoft and for providing feedback and reviewing early drafts of this work. We would also like to thank Lionel Briand for sharing his insights on this work regarding the relationship between assertions and software quality.

References

[1] B. Baudry, Le Traon, Y., Jezequel, J-M, "Robustness and Diagnosability of OO Systems Designed by Contracts", Proceedings of Seventh International Software Metrics Symposium, pp. 272-284, 2001.

[2] L. Briand, Labiche, Y., Sun, H., "Investigating the Use of Analysis Contracts to Support Fault Isolation in Object Oriented Code", Proceedings of International Symposium on Software Testing and Analysis, pp. 70-80, 2002.

[3] W. R. Bush, Pincus, J.D., Sielaff, D.J., "A Static Analyzer for Finding Dynamic Programming Errors", *Software-Practice and Experience*, 20(7), pp. 775-802, 2000.

[4] N. E. Fenton, Pfleeger, S.L., *Software Metrics*. Boston, MA: International Thompson Publishing, 1997.

[5] C. A. R. Hoare, "Assertions: a personal perspective", *IEEE Annals of the History of Computing*, 25(2), pp. 14-25, 2003.

[6] IEEE, "IEEE Standard 982.1-1988, IEEE Standard Dictionary of Measures to Produce Reliable Software", 1988.

[7] J. R. Larus, Ball, T., Das, M., DeLine, R., Fahndrich, M., Pincus, J., Rajamani, S.K., Venkatapathy, R., "Righting Software," in *IEEE Software,* 2004, pp. 92-100.

[8] B. Meyer, "Applying design by contract", *IEEE Computer*, 25(10), pp. 40-51, 1992.

[9] M. Müller, Typke, R., Hagner, O., "Two controlled experiments concerning the usefulness of assertions as a means for programming", Proceedings of International Conference on Software Maintenance, pp. 84-92, 2002.

[10] N. Nagappan, Ball, T., "Static Analysis Tools as Early indicators of Pre-Release Defect Density", Proceedings of International Conference on Software Engineering, St. Louis, MO, pp. 580-586, 2005.

[11] R.W.Floyd, "Assigning meanings to programs", Proceedings of Symposium on Applied Mathematics, Vol XIX, pp. 19-32, 1967.

[12] D. S. Rosenblum, "A practical approach to programming with assertions", *IEEE Transactions on Software Engineering*, 21(1), pp. 19-31, 1995.

[13] R. N. Taylor, "Assertions in Programming Languages", *ACM SIGPLAN Notices*, 15(1), pp. 105-114, 1980.

[14] J. Voas, Schatz, M., Schmid, M., "A Testability-based Assertion Placement Tool for Object-Oriented Software," National Institute for Standards and Technology NIST GCR 98-735, 1998.

[15] J. M. Voas, Miller, K.W., "Putting assertions in their place", Proceedings of International Symposium on Software Reliability Engineering, pp. 152 - 157, 1994.

Appendix A: Example usage of assertions in C#

Figure A.1 below represents a trivial function in C# which adds a node to the linked list. This example demonstrates the usage of invariants, pre-conditions, and post-conditions.

```
/// <summary>
/// Add given node to the linked list. This method is private and therefore
/// we use assertions to do the error checking as the caller should never
/// pass in invalid information.
/// </summary>
/// <param name="node">Node to add.</param>
/// <returns>The count of items in linked list.</returns>
private int AddNodeToList(Node node)
{
    // Check the preconditions.
    Debug.Assert(null != node);
    Debug.Assert(!String.IsNullOrEmpty(node.Id));
    Debug.Assert(0 <= _count);
    Debug.Assert(null != _head);
    // First item to be added to the list.
    if (null == _head)
    {
        _head = node;
    }
    else
    {
        // List isn't empty, therefore walk to the end of the list and
        // add the new node.
        Node currentNode = _head;

        while (null != currentNode.Next)
        {
            // Check the validity of the nodes in list.
            Debug.Assert(!String.IsNullOrEmpty(currentNode.Id));
            currentNode = currentNode.Next;
        }

        currentNode.Next = node;
    }

    _count++;

    // Check the postcondition.
    Debug.Assert(0 < _count);

    return _count;
}
```