Experimental Evaluation of Static Source Code Analysis tools

Khalid Alemerien
Department of Computer Science
North Dakota State University
Fargo, ND, USA
Khalid.alemerien@my.ndsu.edu

Kenneth Magel
Department of Computer Science
North Dakota State University
Fargo, ND, USA
Kenneth.Magel@my.ndsu.edu

Abstract - Code analysis is a substantial process to understand the source code. This needs effective, reliable, and accurate code analysis tools, but these tools may mislead the software developers because they might provide inaccurate measures. Therefore, there is a need to investigate empirically the features of these tools. This paper highlights the serious need to improve understanding of source code to support the development of reliable software in addition to achieve better understanding of how code analysis tools work. For this purpose, this paper presents an experiment, which is comparing between two static code metrics analysis tools. This paper provides significant evidence about the inconsistent values of metrics that are calculated by two code analysis tools for a given program. In addition, our paper shows how the tools are significantly different in terms of speed. Then, this paper discusses numerous of issues and causes of this difference such as unclear definition of code metrics.

Keywords: static source code analysis, software metrics, code analysis tools, reliability, Measurement.

I. INTRODUCTION

The levels of quality, testability, and maintainability of software programs can be improved and measured through utilization of code analysis tools throughout the software development process. In the software development process [1][2][3], code metrics provide appropriate quantitative information about various aspects of software. Therefore, this information supports decision-making in different situations e.g. we can estimate how many test cases we need to cover each piece of code or whether a particular method is complex or not. In both situations, we need to measure the complexity of each method to determine if we need to divide that method to simple methods … and so forth. Therefore, the code analysis tools help to collect metrics from source code or during program runtime.

Basically, there are two kinds of code analysis: Static analysis and dynamic analysis [4]. On one hand, the static code analysis calculates the metrics based on source code without execution, such as the number of lines of code, number of classes, cyclomatic complexity, and so forth, but on the other hand, the dynamic code analysis calculates the metrics during the run-time of the program such as code coverage metrics. In this experiment, we focus on the static code analysis for Java and C++ programming languages. Moreover, a variety of static analysis approaches have been incorporated into open source and commercial tools. Nowadays, there are many static code analysis tools, which are available as open source, freeware, and commercial such as Understand (commercial) [5] and SourceMonitor (freeware) [6]. This pool of tools makes a challenge for developers to decide which tool is better in terms of efficiency, comprehensibility, consistency, and accuracy. Since inadequate data have been published on the actual performance of these tools, there is no certainty that the output of the analysis process, using these tools, is accurate and precise. This leads us to ask the following question: To which degree code analysis tools are reliable to use during the software development process?

Mainly, these tools are widely used to improve overall the quality of decision making, but these decisions might be made based on unreliable or inaccurate data that are provided by code analysis tools. For that reason, there is a need to find evidence or proof to determine whether these tools are reliable or not, as well as explain why this difference exists.

In order to answer these inquiries and illustrate why there are differences between code analysis tools, we designed and performed a controlled experiment examining whether the metrics values collected by code metrics analysis tools are consistent, reliable, and accurate for Java and C++ programs or not. Moreover, we examine the efficiency of code analysis tools in terms of completion time needed to perform the code analysis process. In our experiment, we used Understand and SourceMonitor tools to analyze six object programs with two versions for each one.

Our findings show that software developers must be very careful when choosing any tool to help them comprehend the source code through understanding of how these tools calculate the metrics and grasp the metric definitions accurately. Moreover, our study assists developers with gaining insight into the static source code analysis tools through detailed quantitative evaluation. To our knowledge,
we can say that this paper is the first to discuss this issue, which provides very important information to prove there is a difference between the results of these tools.

The rest of this paper is organized as the following. Section 2 presents related work. Section 3 describes the code metrics analysis tools. Section 4 presents our experiment design and results. Section 5 shows our results, and Section 6 presents conclusions.

II. RELATED WORK

Static code analysis is analysis of the source code, which is performed without actual code execution. Manually, it is a hard and time-consuming process to analyze the source code. Thus, we need automatic approaches to make the static analysis easier and less costly. In addition, the static analysis could be used during the software development process to improve the quality of code. Basically, a number of research studies have attempted to investigate the benefits, features, and performance of code analysis tools. Gomez et al. [4] evaluated the features of five static analysis tools in terms of buffer overflow. This study found that it is important to present the results in understandable manner as well as to provide data about the rules set that the tool enforces. Moreover, Zitser et al. [7, 8] evaluated five static analysis tools, presenting how to effectively detect buffer overflow in C programs. It is important to note that the Hoper’s evaluation [9] focused on security aspects for 30 static analysis tools. Moreover, Emanuelsson and Nilsson [10] surveyed research articles, tools’ manuals, and defect reports in order to identify the functionality provided by these tools. Furthermore, Ernst [11] compared, theoretically, static and dynamic techniques and he observed that there is a need for hybrid technique which incorporates the dynamic and static techniques in single tool.

There are research studies that evaluated the static analysis tools, but unfortunately they do not study the consistency and accuracy of results of the static analysis tools for specific object programs. To this end, we conducted a controlled experiment to evaluate the consistency and accuracy of metrics results of different static analysis tools for the same object programs. Therefore, the principle goal of our experiment is to apply different static code analysis tools to a set of open source programs. And then, we analyzed statistically the results, with focusing on our findings that provide software developers evidence whether the static analysis tools are accurate and consistent or not.

III. DESCRIPTION OF CODE ANALYSIS TOOLS

In order to be able to use the static analysis tools in effective way, we basically performed a comprehensive analysis of static analysis tools. Therefore, among various static code analysis tools [12], we have selected two tools to perform the static code analysis. These tools are Understand and SourceMonitor. Briefly, the main reasons behind choosing these tools are:
- These tools are used for analyzing the source code for both C++ and Java programming languages.
- These tools, basically, follow the same steps for performing the code analysis process.
- These tools are stand-alone applications, which mean you do not need the programming environment to make the code analysis process.
- Understand tool is a commercial tool and SourceMonitor is a freeware tool, which represents different development environments commercial and freeware.
- These tools can be used for analyzing different sizes of code.

A. Understand Tool

We have used Understand version 3.1. It is a commercial code analysis tool. It has many features such as: First, it is a cross-platform, which is used for different operating systems. Second, it supports 17 programming languages in different versions and or compilers. Third, it measures more than 50 metrics for statement, function/method, class, file, and project level. Fourth, it provides over 20 different graphs. Fifth, it shows the dependencies of code pieces. Finally, it generates a variety of output reports.

B. SourceMonitor Tool

We have used SourceMonitor version 3.3. It a freeware application, which is used to measure code metrics in terms of the number of lines of code, number of files, number of classes, number of functions, and methods. In addition, it helps to identify the relative complexity of methods/functions based on Cyclomatic Complexity (CC) and modified version of CC. Moreover, it measures the code metrics for source code written in Java, C, C++, C#, VB.NET, and others.

IV. THE EXPERIMENT

We wish to address the following research questions:

**RQ1:** Is the SourceMonitor tool more efficient than Understand tool in terms of completion time of the code analysis process.

**RQ2:** Are the values of metrics measured by Understand tool inconsistent with the values of metrics measured by SourceMonitor tool for a given program.

In order to address our research questions, we designed a controlled experiment. The following subtitles present, our object programs, independent variables, dependent variables and measures, experiment design, threats to validity, and data and analysis.

A. Object Programs

In our experiment, we used two programming languages, Java and C++ to investigate the features of code analysis tools such as pointing out the impact of programming
language on the effectiveness of code analysis tools. We conducted our experiment on a variety of object programs in order to make our findings as generally representative as possible. The following existing software projects were used in our experiment. The C++ programs are 3Depict, CppCMS, and Thunderbird while the Java programs are Jtopas, Apache_Ivy, and ApacheMeter. We have chosen these to represent diversity of development contexts, programming languages, and code source size.

Apache_Ivy – is a popular dependency manager as well as it is a sub-project of the Apache Ant Project [13].
ApacheJmeter – is an open source software, which is developed to load functional tests behavior and measure the performance of static and dynamic resources [14].
Jtopas – is a Java library, which is used for the common problems of parsing text data [15].
Thunderbird – is a free open source email and news client application developed by the Mozilla Foundation [16].
CppCMS – is an open source web application framework for the C++ programming language. It is used for Rapid Web Application Development [17].
3Depict – is an open source software for analysis of scientific datasets as well as a visualization application [18].

Table 1 shows, for each object program, a “Version” (the version numbers), “Programming Language” which is used for developing the program, “Size” (Number of Classes).

Basically, we classified our objects based on the number classes into three categories Large (>=1001 classes), Medium (501-1000 classes), and Small (<=500 classes).

B. Variables and Measures

1. Independent Variables

Our experiment manipulated one independent variable: A code metric tool. We used two code analysis tools, Understand (Commercial) and SourceMonitor (Freeware) tools, which represent two different software development environments.

2. Dependent Variables and measures

- **Completion time of the code analysis process**
  To investigate our research questions we need to measure the efficiency of code analysis tools in terms of speed. Therefore, we use a metric, time of completion for the code analysis process for a given program.

- **Variance in Metrics values**
  For each version of program, we collected the following metrics using both Understand and SourceMonitor tools: First, the number of lines of code, number of statements, number of functions (for C++), and number of methods (for Java), Second, percentage of comments to lines of code, Third, the maximum complexity and maximum depth of inheritance for overall functions/methods, Finally, the average complexity of functions for C++ or methods for Java.

C. Experiment Design

There were two types of data to be collected among this experiment: Time of completion the code analysis process and code metrics. Therefore, we obtained completion time of the code analysis process by running six object programs with two different versions for each program on both static analysis tools, Understand and SourceMonitor. Thus, we ran each version of program two times, one with Understand and another with SourceMonitor. For each version of program, we collected the following metrics using both Understand and SourceMonitor tools: First, the number of lines of code, number of statements, and number of functions (for C++ and methods (for Java), Second, percentage of comments to lines of code, Third, the maximum complexity of functions for C++ or methods for Java.
complexity and maximum depth of inheritance for overall functions/methods, and finally, the average complexity of functions/methods. These metrics help us to investigate the impact of programming language, size of program, and the code analysis tools on the consistency of the output of code analysis tools.

The following steps demonstrate the overview of our experimental procedure:
1 – Run the code analysis tool
2 – Upload the object program.
3- Configure the tool settings for the object program.
4 – Choose the metrics that we need to measure.
5 - Repeat the steps 1-4 for all versions.

After the completion these steps that we described above, we obtained 24 log files which contains both completion time of the analysis process and the values of static metrics from both Understand and SourceMonitor tools for all program versions.

D. Threats to Validity

1. Internal Validity

First, the findings that we have obtained about the efficiency of code metrics analysis tools could be affected by potential faults in completion time calculations in our experiment tools. We addressed this threat by executing the tools on various sizes of C++ and Java programs. Second, the conclusions about the consistency of the values of code metric could be affected by unclear definition of each metric that was collected by analysis tools. To control this threat we reviewed carefully the definitions provided by the developers of tools for metrics and, moreover, we specified only the metrics which have the same definition in both analysis tools.

2. External Validity

The conclusions about the efficiency of code metrics analysis tools could be effected by this factor: Time calculation by the tools might be affected by execution of other system processes on the same machine that we used in our experiment. To address this threat we executed the analysis process more than one time in addition to we recorded the time manually.

E. Data Analysis

At the first, our hypothesis associated with RQ1 is: (H1) the SourceMonitor analysis tool is faster than Understand analysis tool. Also, the hypothesis associated with RQ2 is: (H2) the code metrics that are measured by both Understand and SourceMonitor tools are different.

We used ANOVA test to calculate the significance p. We used the R, it is a programming language, to perform statistical analysis and we used the BoxPlotter [19], it is an online tool, for drawing the Boxplots.

This section presents the collected data for all programs. We used Boxplots to show the results of the eight code metrics. Each boxplot contains two boxes showing the distribution of measurement scores for each of the two tools, across each of the versions of the object programs. The above box represents the scores of Understand tool while the lower box represents the scores of SourceMonitor tool.

So, we used the completion time of the code analysis process to accept/reject our hypotheses associated with RQ1 as well as we used the rest of metrics to accept/reject our hypothesis associated with RQ2. For each metric, this paper provides a Boxplot, which shows an overview of the collected data. Therefore, we present our research questions with their analysis granularity as the following.

1. **RQ1: Efficiency of Code Analysis Tools**

Our first research question assumes that the SourceMonitor tool is more efficient than Understand tool. For evaluating the efficiency of code analysis tools, we used the speed of tools in terms completion time as a measure of tool’s efficiency. To test this hypothesis, we performed ANOVA test (df=1) for each tool per program, at a significance level (<0.001). Our findings show a significant difference between the tools with (p=0.0004547). Therefore, our hypothesis (H1) is supported. Fig. 1 shows the difference between the mediums of completion time of the two tools. To sum up, the SourceMonitor is more efficient than the Understand, as we previously assumed.

2. **RQ2: Output Consistency of Code Analysis Tools**

Our second research question assumes that there are differences between the values of metrics that are measured by SourceMonitor and Understand. For evaluating the consistency of values of the code metrics, we used seven metrics as the measures of output tools’ consistency. To test this hypothesis, we performed ANOVA test (df=1) for each
tool per program, at a significance level (<0.001). Our results are shown in the following subsections:

**Number of lines**

It is a code metric, which is used as a measure of program size. Boxplot in Fig. 2 shows the difference between the mediums of the two tools in terms of the number of lines for all programs. Statistically, from the ANOVA test, there is a significant difference between two tools with \( p=2.2\times10^{-16} \). Therefore, our hypothesis \( (H2) \) is supported in terms of number of lines. To sum up, the results for each SourceMonitor and Understand are significantly different in terms of number of lines, as we previously assumed.

![](image1.png)

Fig. 2. Total number of lines for all programs

**Number of statements**

It is a code metric, which is used as a measure of program size as well. Boxplot in Fig. 3 shows the difference between the mediums of the two tools in terms of the number of statements for all programs. Statistically, from the ANOVA test, there is a significant difference between two tools with \( p=1.2\times10^{-16} \). So, our hypothesis \( (H2) \) is supported in terms of number of statements. To sum up, the results for each SourceMonitor and Understand are different in terms of number of statements, as we previously assumed.

![](image2.png)

Fig. 3. Total number of statements for all programs

**Number of functions (for C++) and methods (for Java)**

It is a code metric, which is used as a measure of program size as well. Boxplot in Fig. 4 shows the difference between the mediums of the two tools in terms of the number of functions for all programs. Statistically, from the ANOVA test, there is a significant difference between two tools with \( p=0.009386 \) at significant level (<0.01). Therefore, our hypothesis \( (H2) \) is supported in terms of number of functions. To sum up, the results for each SourceMonitor and Understand are significantly different in terms of number of functions, as we previously assumed.

![](image3.png)

Fig. 4. Total number of functions for all programs

**Percentage of comments to lines of code**

It is a code metric, which is used as a measure of program size and readability as well. Boxplot in Fig. 5 shows the difference between the mediums of the two tools in terms of the percentage of comments to lines of code for all programs. Statistically, from the ANOVA test, there is a significant difference between two tools with \( p=8.066\times10^{-7} \). Therefore, our hypothesis \( (H2) \) is supported in terms of percentage of comments to lines code. To sum up, the results for each SourceMonitor and Understand are significantly different in terms of percentage of comments to lines of code, as we previously assumed.

![](image4.png)

Fig. 5. Percentage of comments to lines of code for all programs
Maximum complexity

It is a code metric, which is used as a measure of the complexity of program. Boxplot in Fig. 6 shows the difference between the mediums of the two tools in terms of the maximum complexity for all programs. Statistically, from the ANOVA test, there is a significant difference between two tools with \((p=4.92\times10^{-7})\). Therefore, our hypothesis \((H2)\) is supported in terms of maximum complexity. To sum up, the results for each SourceMonitor and Understand are significantly different in terms of maximum complexity, as we previously assumed.

![Fig. 6. The maximum complexity of functions/methods for all programs](image)

Average complexity of functions/methods

It is a code metric, which is used as a measure the complexity of program as well. Boxplot in Fig. 8 shows the difference between the mediums of the two tools in terms of the average complexity of functions/methods for all programs. Statistically, from the ANOVA test, there is a significant difference between two tools with \((p=0.000149)\). Therefore, our hypothesis \((H2)\) is supported in terms of the average complexity of functions/methods. To sum up, the results for each SourceMonitor and Understand are significantly different in terms of the average complexity of functions/methods, as we previously assumed.

![Fig. 8. The average complexity for all programs](image)

Maximum depth of inheritance for overall functions/methods

It is a code metric, which is used as a measure of program complexity as well. Boxplot in Fig. 7 shows the difference between the mediums of the two tools in terms of the maximum depth of inheritance for each function or method cross programs. Statistically, from the ANOVA test, there is a significant difference between two tools with \((p=0.0003803)\). Therefore, our hypothesis \((H2)\) is supported in terms of maximum depth of inheritance. To sum up, the results for each SourceMonitor and Understand are significantly different in terms of maximum depth of inheritance, as we previously assumed.

![Fig. 7. The maximum depth of inheritance for all programs](image)

To summarize this, all the observations for the seven code metrics in this experiment support our second hypothesis, which assumes that the values of metrics are significantly different.

V. DISCUSSION

Our results strongly support the conclusion that the values of code metric, were calculated by the code analysis tools, are different. This means some or all these values might be calculated in wrong way. Moreover, we attempt to demonstrate causes of these differences between the two tools as the following:

- Unclear definition of metrics: For some metrics, there are trivial differences in definition of metrics. Therefore, this kind of difference might cause different calculations, which leads the difference of metrics values between the two tools. For example, SourceMonitor [6] defines number of statements as "In C++, computational statements are terminated with a semicolon character. Branches such as if, for, while and goto are also counted as statements. The exception control statements try and catch are also counted as statements. Preprocessor directives #include, #define, and #undef are counted as statements. All other preprocessor directives are ignored. In addition all statements between each #else or #elif statement and its
closing #endif statement are ignored, to eliminate fractured block structures.” By contrast, Understand [5] defines number of statements as “Number of declarative plus executable statements.”

- Errors in calculation of metrics: It might be exist in any of them. Consequently, we may not be able to say that, but we expect that. For example, the value of maximum depth of inheritance metric that was calculated by SourceMonitor is 9+ for all Java and C++ programs. By contrast, the values of maximum depth of inheritance metric using Understand are varying for object programs (1, 2, 5, 6, 9, and 10).

- Programming language: We noticed that the metrics were calculated for Java programs; have slight differences comparing with metrics of C++ programs for both tools. For example, the number of functions is significant different using both tools for any given C++ program in our experiment. By contrast, the number of functions is slight different using both tools for any given Java program in our experiment

- Structure organization of program: The analysis process may depend on how much the object program structurally organized.

- Different preprocessing steps: These steps might be varying from tool to another. For that, it might effect on the effectiveness of tool as well as the calculations.

Moreover, the SourceMonitor tool is more efficient than Understand tool.

Consequently, this difference among the results makes the code analysis tools unreliable enough. Therefore, the decision making under uncertainty, it needs more investigation about which tool is better. Moreover, decision makers have to specify the circumstances needed to choose the accurate tool.

VI. CONCLUSIONS

We have presented our study of two code analysis tools, which were used to analyze the source code of six programs in both C++ and Java programming languages. Furthermore, we used two versions per program. We found that there is a strong significant difference among the values of code metrics between two tools. Also we found that the SourceMonitor tool is more efficient than the Understand tool. Through the results are reported in this paper, we hope to provide valuable findings for practitioners, especially the developers of code analysis tools as well as the software developers in general.

For future work, we plan to conduct wider study that involves more tools and object programs. Also, we will attempt to examine the effectiveness of using various visualization techniques with code analysis tools. Furthermore, we plan to perform further studies to investigate analysis tools with different versions in order to examine the stability of these tools.

REFERENCES