Preprint of paper published in Proc. International Conference on Software Testing, Verification, and Validation (ICST 2009), pp. 141-150, 2009.

The Effectiveness of Automated Static Analysis Tools for Fault Detection and Refactoring Prediction

Fadi Wedyan, Dalal Alrmuny, and James M. Bieman
Colorado State University
Computer Science Department
Fort Collins, CO 80523-1873
{wedyan, alrmuny, bieman}@cs.colostate.edu

Abstract

Many automated static analysis (ASA) tools have been developed in recent years for detecting software anomalies. The aim of these tools is to help developers to eliminate software defects at early stages and produce more reliable software at a lower cost. Determining the effectiveness of ASA tools requires empirical evaluation. This study evaluates coding concerns reported by three ASA tools on two open source software (OSS) projects with respect to two types of modifications performed in the studied software CVS repositories: corrections of faults that caused failures, and refactoring modifications. The results show that fewer than 3% of the detected faults correspond to the coding concerns reported by the ASA tools. ASA tools were more effective in identifying refactoring modifications and corresponded to about 71% of them. More than 96% of the coding concerns were false positives that do not relate to any fault or refactoring modification.

1 Introduction

Automated static analysis (ASA) tools can identify a wide range of probable software anomalies such as non-compliance with coding and documentation standards, dead code and unused data, security leaks, null pointer dereferencing, endless loops, and floating-point arithmetic problems [1, 9, 17].

Identifying potential causes of failures at an early stage enables developers to solve problems before they are reported during testing or by users. However, even though many of the ASA tools are free, these tools have not achieved the success expected by their designers. ASA tools are usually described by their designers as fault detection, bug finding, error-detection, or anomaly detection tools. Munger et al. [14] call the reported potential defects *coding concerns*.

In this paper, we evaluate the usefulness of ASA tools for Java code. Our goal is to determine the capabilities and costs of using these tools to pinpoint real defects in the software, as well as needed refactorings. We consider the following four research questions:

1. How effective are ASA tools in detecting faults?
2. How effective are ASA tools in predicting code refactoring modifications?
3. What proportion of the reported coding concerns are not real faults or do not represent eventual refactorings?
4. How many faults or eventual refactorings are not detected by ASA tools?

To answer these questions, we conducted a retrospective case study of two successful evolving OSS systems. As far as we know, developers of the studied systems did not use ASA tools in the design and implementation process. The case study considers the following hypothetical situation:

> In an alternative universe, developers use ASA tools to develop an early version of a system. The ASA tools identify potential faults and constructs that possibly should be refactored. The developers evaluate the results of the ASA tools to determine which of the reported concerns represent real faults or needed refactorings. Then they make the changes that they decide are really needed.

To answer the four research questions with respect to the hypothetical situation, we apply the tools to an early version of these systems and determine whether or not the concerns identified by the ASA tools represent either faults that are

eventually repaired or constructs that are eventually refactored. We assume that the hypothetical developers would have immediately made changes in response to the concerns flagged by the tools. However, we assume that the hypothetical developers would only make the same changes that were eventually made to subsequent versions in the real world development. The rationale is that these actual modifications represent concerns that were worth a response. The hypothetical development would benefit from earlier correction of faults and earlier refactoring, with the cost of using the ASA tools. A potential cost of tool use is the effort required to determine if a reported concern is serious enough to warrant a correction or refactoring.

Both of the studied OSS systems are written in Java: *jEdit* [8], a program text editor, and *iText* [13], a Java library for creating PDF, HTML, RTF, and XML documents. There is no evidence that ASA tools were used to develop these systems. We applied three ASA tools (FindBugs [17], IntelliJ IDEA [9], and Jlint [1]) to 13 releases of *iText* and seven releases of *jEdit*. For each release, we identified two types of modifications performed during the release development: *faults fixes* and *refactorings*. Faults fixes are modifications made in response to a detected fault. The fault is the cause of a failure reported by users or detected by testers. Refactorings include modifications aimed to make the code faster or more efficient, modifications of documentation, and the removal of unused data or code. Actual fault fixes and refactorings were compared to the coding concerns reported by the ASA tools to see whether the changes correspond to the reported concerns. In such cases, we say that a fault was detected or a refactoring was predicted by the ASA tool. Concerns that do not relate to any of the corrections are classified as potential *false positives* while corrections or refactorings that were not related to any of the concerns are *false negatives*.

The rest of this paper is organized as follows: Section 2 reviews related work. Section 3 describes the methods used to conduct the study. Section 4 presents the results. Section 5 discusses the threats to validity. Finally, Section 6 gives the conclusions and describes potential future work.

2 Related Work

Few studies have been performed to evaluate the usefulness of ASA tools. Munger et al. [14] investigated coding concerns in object oriented commercial and OSS systems and recognized the need for empirical studies to investigate links between concerns and faults in software systems. They compared commercial software with OSS in order to evaluate claims that OSS code is of high quality. Their results showed that *NetBeans* exhibits a similar density of coding concerns as that of two commercial systems.

Ruter et al. compared ASA tools for Java software [19]. They studied five tools: *Bandera*, *ESC/Java*, *FindBugs*, *Jlint*, and *PMD*. They tested five mid-sized programs and compared overlapping between the reported concerns by each tool on each fault category. The tools agree on some of the reported concerns but generally find many non-overlapping concerns. However, the study did not determine whether reported concerns identify real faults.

Nagappan and Ball [15] showed that there is a strong correlation between defect density determined by ASA tools and defect density determined during the testing phase before release. The study found that ASA tools can distinguish between high and low quality components. However, it did not show whether the reported coding concerns are related to the defects reported by testing.

Kothari et al. [12] developed tools to detect coding concerns as part of a framework that embodies a pattern-based approach, and developed a domain-specific inspection tool. Customization, visualization and reporting capabilities increases the usability of the tools.

Zheng et al. [23] evaluated the economic value of ASA tools in the software development process. They examined software produced by a company that requires the use of ASA tools as part of their development process, and compared the performance of ASA with two defect removal practices: *inspection* and *testing*. ASA cost and defect finding efficiency were similar to that of inspection but ASA was less efficient in finding defects than testing. ASA tools were most effective in detecting two types of defects: *assignment* and *checking*. These are categorized according to the IBM Orthogonal Defect Classification [5], where the *assignment* type includes errors such as initialization and data structures and the *checking* type includes program logic that fails to validate data before use. Zheng et al. did not demonstrate that the use of ASA tools in the development process produced higher quality software. The computation of the cost of using ASA included the money cost of prescreening, but not the time cost. The number of coding concerns reported by most ASA tools is high and manually separating false positives from true defects is a time consuming and error prone process [18].

To the best of our knowledge, this is the first study that evaluates ASA tools by mapping real faults to coding concerns reported by the ASA tools. The results will be useful for both developers and ASA tools designers. Knowing the limitations of ASA tools will identify specific needs for improments of the tools.

3 Study Method

Achieving the research goals required us to

1. develop metrics to answer the research questions,
2. select OSS projects to study,

3. choose ASA tools for code inspection, and

4. develop a methodology to map the coding concerns to the detected faults, and detected refactorings.

The following subsections describe the research questions and the metrics used to answer them, the studied systems, code inspection tools, sources of failure data and the mapping process used in this study.

3.1 Research questions and evaluation metrics

In order to evaluate the performance of the ASA tools, we applied the Goal-Question-Metric paradigm [2, 3]. The research goal and questions are given in the introduction. The following metrics can help to answer the research questions:

1. Fault Detection Ratio.

   - Q1: How effective are ASA tools in detecting faults?
   - Metrics: number of fault fixes performed in a software release, number of reported coding concerns that correspond to the fault fixes.

   The *detection ratio* (1) indicates the proportion of the faults that are detected using ASA tools. This measure is similar to the *defect removal efficiency* defined by Zheng et al. [23] as "the percentage of total bugs eliminated before the release of a product". However, since we are performing the study on released software, the defect removal efficiency cannot be evaluated. Moreover, we can only count the number of faults reported and fixed after release due to failures. Our detection ratio uses the number of fault fixes found in the CVS repository. Some faults are detected by the tools in many releases, but are fixed in a later release. These faults are counted as one detection, which is used to compute the fault detection ratio for the release in which the fault is fixed.

$$\text{Detection Ratio} = \frac{\text{No. Fixed faults Detected by ASA}}{\text{Total no. faults fixed}} \quad (1)$$

2. Refactoring Ratio.

   - Q2: How effective are ASA tools in predicting code refactorings?
   - Metrics: number of refactoring modifications performed, number of reported coding concerns that correspond to performed refactorings.

   The *refactoring ratio* (2) measures how effective the ASA tools are in finding code segments that are later modified to improve a design. Refactoring changes are all modifications that aim to "clean up the code" but do not solve any software failure or add new functionality. These modifications include code organization, changes to documentation, removing unused data and code, simplification of logical and mathematical expressions, and better utilization of memory and computational resources. Refactorings are identified by inspecting successive versions for relevant changes.

$$\text{Refactoring Ratio} = \frac{\text{No. performed refactorings recommended by ASA}}{\text{Total no. refactorings performed}} \quad (2)$$

3. False Positive Ratio.

   - Q3: What proportion of the reported coding concerns are not real faults or performed refactorings?
   - Metrics: number of false positives, number of coding concerns.

   We computed *false positive ratios* (3) related to (a) faults, (b) refactorings, and (c) both faults and refactorings. If more than one coding concern map to one fault fix or refactoring, then all of these concerns are not counted as false positives. We encountered few of these cases and all were in the refactoring category.

$$\text{False Positive Ratio} = \frac{\text{No. False Positives}}{\text{No. coding concerns}} \quad (3)$$

   Some false positives might indicate real faults, which developers did not correct because they were not reported or were overlooked. Because we are not able to identify the real faults among the potential false positives, we treat all potential false positives as if they do not identify real faults — potential false positives are treated as actual false positives.

4. False Negative Ratio

   - Q4: How many faults or performed refactorings are not detected by ASA tools?
   - Metrics: number of false negatives, number of modifications.

   False negatives (4) are faults fixes and refactorings performed on the code but not detected by the ASA tools. In a manner similar to the computation of false positives, there are three false negatives ratios: one for

Table 1. Main characteristics of the studied software.

Project	JEdit	iText
Releases	26	49
Developers	138	15
Downloads	3,693,141	192,054
KLOC	140.6	99.8
No. Files	394	685

faults, one for refactorings, and one for both faults and refactorings.

$$\text{False Negative Ratio} = \frac{\text{No. False Negatives}}{\text{No. modifications}} \quad (4)$$

In order to compute the overall ratios over all the releases, we ignored concerns that are repeated in subsequent releases.

3.2 Studied Systems

We studied two OSS projects developed in Java, *jEdit* [8] and *iText* [13]. These are two successful projects from SourceForge [20]. The success of these projects is indicated by the number of downloads from SourceForge, the number of beta and stable releases, and the age of the software [6]. Table 1 provides a summary of the main characteristics of the studied systems.

jEdit [8] is a well-known program text editor that supports 130 file types. The project started in 1999 and the development team consists of 138 developers. To keep the study effort within our resources, we examined only the Java files in the core source code package. We excluded all plug-ins and non-Java source code.

iText [13] is an open source Java library for creating PDF, HTML, RTF, and XML documents. The iText project started in late 2000, and is quite popular. The key capability of iText is its ability to produce PDF files dynamically and for free [7].

Due to time and resource limitations, the study included only the first 13 releases of *iText* and the first seven releases of *jEdit*.

3.3 Static Analysis Tools

Several tools are available for detecting coding concerns. These tools are either commercial, open source, or developed by researchers. The tools vary in the number and the type of concerns they can detect as well as the programming languages they support. This study evaluated three ASA tools: IntelliJ® IDEA, Jlint, and FindBugs.

3.3.1 IntelliJ® IDEA

IntelliJ IDEA [9] is a commercial comprehensive development environment that provides enhanced development tools including a code inspection tool. IntelliJ, developed by the JetBrains company, inspects code looking for 632 coding concerns organized in 49 groups. We obtained a 30-day trial version of IntelliJ IDEA.

3.3.2 FindBugs

FindBugs [17] is a free ASA tool developed at the University of Maryland. FindBugs uses static analysis to inspect Java bytecode for occurrences of potential faults, and maps these faults to the corresponding Java source code. It is written in Java, and can be run with any virtual machine compatible with Sun's JDK 1.4 and can analyze programs written for any version of Java. The tool website [17] reports that the ratio of false warnings is below 50%.

3.3.3 Jlint

Like FindBugs, Jlint [1] is a free ASA tool that analyzes Java Bytecode. Jlint performs syntactic checks and Data flow analysis. Jlint also detects synchronization problems by building a lock graph and verifying that the lock graph is cycle free. Like FindBugs, Jlint reports concerns in terms of Java source code.

3.4 Data collection and mapping concerns to faults fixes and refactorings

We obtained the necessary fault data from the CVS repositories and the failure report databases of the studied projects. In SourceForge, users report failures or what they consider to be failures. Developers review the failure reports and decide whether or not a reported failure needs to be fixed. The developer either fixes the fault that caused the failure and modifies its status, or provides an explanation of why it is not a real failure.

The fault data indicates that developers have determined that most reported failures are not real failures and do not need to be fixed. Some reported failures represent a misunderstanding of the software by reporting users. Many of the reported failures are caused by the installation process, which may be due to poor documentation and installation guidelines.

CVS tracks modifications made by the developers along with the developers' comments. CVS also traces the changes made on the code between the different updates.

For every release of the studied software, we traced the updates made on every Java file on that release until the next release, including all intermediate releases. We reviewed the comments added by the developers to determine whether the update was a failure fix or not. The analysis did not depend on comments. If there were no comments on an update, we examined the changes made to determine whether it was a failure fix or a different type of change, i.e., adding an attribute, changing the code structure, etc. We studied only failures with closed status. A closed failure is one that was handled by a developer either by providing a comment or fixing the failure.

By examining the updates in the CVS repositories, we found that many code changes were not done in response to entries in the failure report database. These changes might represent faults that caused failures that were detected by the developers, or were reported to the developers by other means, for example, by email. They may also represent faults that have not yet caused failures that have been observed. We then examined the concerns reported by the ASA tools to see whether they correspond to the performed modification.

4 Results

Tables 2 and 3 summarize the coding concerns reported by the ASA tools on iText and jEdit, respectively. Note that the totals given in the last rows of the tables represent the total number of unique concerns.

None of the coding concerns reported by *Jlint* correspond to any of the faults or refactorings detected in the studied systems. To simplify the presentation, we do not include results from *Jlint* in the tables.

We use the data collected on *iText*, *jEdit*, and *Jlint* along with the metrics developed in Section 3.1 to answer the four questions posed in Section 1. Section 4.1 gives the results of mapping concerns to the faults and refactorings in each of the studied systems. Sections 4.2 through 4.5 evaluate the answers to each of the four questions and associated measures.

4.1 Mapping coding concerns to modifications

Table 4 shows the fault corrections and refactoring actions performed on each release of the studied seven releases of *jEdit* (columns two and five, respectively). Columns three and four show the number of corresponding coding concerns identified by the two ASA tools that map to actual fault corrections while columns six and seven show the corresponding concerns identified by the tool that map to actual refactorings. Table 5 displays the corresponding results for *iText*.

Table 2. Reported faults and refactorings in 13 releases of iText identified by FindBugs and IDEA. Totals include only unique concerns.

iText	Faults		Refactoring	
Release	FindBugs	IDEA	FindBugs	IDEA
0.87	151	403	67	3847
0.9	154	407	66	3868
0.91	169	412	69	3913
0.92	169	414	69	3919
0.93	164	412	65	3914
0.93b	164	412	65	3914
0.94	229	434	65	4019
0.95	166	429	66	4160
0.96	166	457	72	4191
0.97	180	486	78	4213
0.98	196	484	66	4224
0.99	191	484	90	4261
1.00	201	492	90	4267
Total	237	530	98	4333

Table 3. Reported faults and refactorings in 7 releases of jEdit identified by FindBugs and IDEA. Totals include only unique concerns.

jEdit	Faults		Refactoring	
Release	FindBugs	IDEA	FindBugs	IDEA
3.0	72	238	38	1491
3.0.1	72	236	38	1493
3.0.2	76	236	37	1498
3.1	102	256	67	1641
3.2	100	319	68	1908
3.2.1	101	320	68	1911
3.2.2	118	319	87	1919
Total	130	341	91	1989

4.2 Fault detection efficiency

The results for these ASA tools shown in Tables 4 and 5 are disappointing. Out of 112 faults that caused failures in *jEdit*, FindBugs was only successful in identifying one fault while IDEA identified three faults. Again, *Jlint* identified none of the faults. Since the fault found by FindBugs was also found by IDEA, the ASA tools combined identified only three out of 136 faults in *iText*.

Both FindBugs and IDEA detected one infinite recursive call in both *jEdit* and *iText*. The tools also detected an incorrect use of a static member in *iText*. FindBugs identified an incorrect implementation of a serializable interface in *iText*. IDEA did not detect that fault. On the other hand, IDEA detected three infinite loops, two in *jEdit* and one in *iText* that FindBugs failed to detect. The ASA tools detected a total of four different faults in *iText*, and three in *jEdit*.

Table 6 summarizes the *detection ratios* obtained by FindBugs and IDEA on *jEdit* and *iText*, and the ratios for using both tools. A total *detection ratio* of 2.42% was achieved by both ASA tools. The best detection ratio of 2.68% was achieved by IDEA on *jEdit*. Since *Jlint* identified none of the faults, its fault detection ratios are zero (not shown in Table 6). The worst non-zero fault detection ratio is 0.9% for FindBugs on the same data set.

The ASA tool documentation indicates that the tools should be able to detect faults related to *null* dereferencing. However, this was not the case. Consider the fault fix shown in Figure 1 which was performed in *iText*. All of tools failed to detect that object *line* can be null at the time it calls method *size()*. We found 12 fault fixes that are null references and none were detected by the ASA tools.

We also found 23 bug fixes that relate to incorrect string operations. These include out-of-index errors, use of *equals* instead of equalsIgnoreCase, use of == in string comparison, etc. Other simple faults undetected by the ASA tools include missing *break* in a *switch ... case* statement, type casting problems, misplaced braces, and initialization problems.

Based upon our data, the computed fault detection ratios were less than 3% even when computed with all three of the ASA tools. That means if the developers had decided to use the ASA tools to detect the causes of the reported failures, the tools would be able to detect fewer than 3% of the faults.

```
before: if (line.size() > 0)
{...}
after: if (line != null&&line.size()>0)
{...}
```

Figure 1. Null dereferencing fault not detected by the ASA tools

Table 6. Fault detection ratios (%)

ASA Tool	iText	jEdit	Both Projects
FindBugs	2.22	0.9	1.62
IDEA	2.22	2.68	2.42
Both Tools	2.94	2.68	2.42

4.3 Refactoring efficiency

Table 7 summarizes the *refactoring ratios* obtained by FindBugs and IDEA on *jEdit* and *iText*, and the ratios for all of the tools. The *refactoring ratios* for Jlint are zero (not shown in Table 7), since *Jlint* identified none of the refactorings. The ratios for all of the tools is the same ratios obtained by IDEA, since the refactorings detected by IDEA subsume the ones detected by FindBugs, and (trivially) those found by Jlint. IDEA was clearly more successful and detected about 79% of the refactorings in jEdit and scored about 71% refactoring ratio on both data sets.

Table 7. Refactoring Ratios (%)

ASA Tool	iText	jEdit	Both Projects
FindBugs	6.93	16.88	10.06
IDEA	65.35	79.21	70.56
Both Tools	65.35	79.21	70.56

IDEA was successful in detecting most of the concerns related to Java documentation, unused variables and data members, redundant casting, declared exceptions that were never thrown, incorrect scope modifier, and unused code. FindBugs detected some of the unused data members and incorrect modifiers for data members. FindBugs does not report concerns for Java documentation, the category in which IDEA was mostly successful.

All of the tools missed many of the optimization operations. For example, using method *isEmpty()* is faster than using the condition *size()==0* to check if a structure is empty. Another example is shown in Figure 2. The developers of *iText* replaced the call to method *size()* in the condition of the *for* loop with a variable that holds the structure size. Such optimizations occurred frequently especially when the developers clean up the code before a release.

```
 int size = kids.size();
 for (int k=0; k < size; ++k)
...
```

Figure 2. An optimization not detected by the ASA tools

Table 4. Fault corrections and refactorings performed in 7 releases of *jEdit* along with the corresponding coding concerns found by FindBugs and IDEA

Release	Fault Corrections	Corrected faults found by FindBugs	Corrected faults found by IDEA	Refactorings Performed	Refactorings correctly predicted by FindBugs	Refactorings correctly predicted by IDEA
3.0	18	0	1	0	0	0
3.0.1	2	0	0	0	0	0
3.0.2	26	1	1	25	6	22
3.1	22	0	1	31	4	24
3.2	2	0	0	0	0	0
3.2.1	3	0	0	0	0	0
3.2.2	39	0	0	21	3	15
Total	112	1	3	77	13	61

Table 5. Fault corrections and refactoring performed in 13 releases of *iText* along with the corresponding coding concerns found by FindBugs and IDEA

Release	Fault Corrections	Corrected faults found by FindBugs	Corrected faults found by IDEA	Refactorings Performed	Refactorings correctly predicted by FindBugs	Refactorings correctly predicted by IDEA
0.87	34	1	1	0	0	0
0.9	3	0	0	2	0	0
0.91	6	0	0	4	3	3
0.92	22	1	1	29	0	26
0.93	6	0	0	16	0	6
0.93b	1	0	0	3	0	1
0.94	5	0	0	2	0	1
0.95	7	0	0	3	0	0
0.96	15	0	0	3	0	2
0.97	5	0	0	0	0	0
0.98	5	1	1	4	0	0
0.99	6	0	0	6	2	1
1.00	21	0	0	29	2	26
Total	136	3	3	101	7	66

Our results indicate that ASA tools can identify some faults before they are found by users. The tools do a better job in predicting refactorings that will be performed to develop later versions of the software. The cost of using the tools includes the effort required to examine many false positives.

4.4 False Positives

Table 8 shows the false positive ratios for FindBugs. The ratios are similar for the faults category in both data sets. In the refactoring category, FindBugs reported fewer false positives in *jEdit* than *iText* and a lower overall ratio than for the faults category in both data sets. IDEA's false positive ratios are shown in Table 9. The false positive ratios for IDEA are higher than for FindBugs with higher ratios in the refactoring category. All of the false positive ratios for Jlint are 100%, since none of reported concerns represent faults or refactorings.

Table 8. FindBugs False Positives (%)

	iText	jEdit	Both Projects
Faults	98.73	98.61	98.70
Refactoring	92.85	65.79	85.29
Faults & Refactorings	97.01	93.67	95.68

Table 9. IDEA False Positives (%)

	iText	jEdit	Both Projects
Faults	99.43	99.12	99.31
Refactoring	98.47	96.93	97.99
Faults & Refactorings	98.58	97.25	98.15

Reporting too many coding concerns has been always a problem with ASA tools [14]. The abundance of false positives will discourage developers from using the tools. Consider the number of coding concerns reported by IDEA (shown in Table 2). If a developer of *iText* decides to find the cause of a failure in release 0.98 with the help of IDEA, he/she must look over 484 reported faults.

4.5 False Negatives

False negative ratios (equation 4) refer to the percent of faults or refactorings not detected by the ASA tools. These ratios are shown for FindBugs and IDEA in tables 10 and 11, respectively. About 98% of the faults were not detected by either FindBugs or IDEA. In the refactoring category, IDEA missed about 21% of the refactorings in *jEdit*, 35% in iText, and about 29% in both data sets. FindBugs was less successful and missed more than 89% of the refactorings in both data sets. Since Jlint missed all of the refactorings and faults, all of the false negative ratios are 100%.

Table 10. FindBugs False Negatives (%)

	iText	jEdit	Both Projects
Faults	97.79	99.11	98.38
Refactoring	93.07	83.12	88.76
Faults & Refactorings	95.78	92.59	94.37

Table 11. IDEA False Negatives (%)

	iText	jEdit	Both Projects
Faults	97.79	97.32	97.58
Refactoring	34.65	20.77	28.65
Faults & Refactorings	70.89	66.13	68.78

5 Threats to Validity

All empirical studies, especially case studies, have limitations [11, 22]. We assess four types of threats to the validity of this empirical study: construct validity, content validity, internal validity and external validity. Construct validity refers to the meaningfulness of measurements [10, 16] — do the measures actually quantify what we want them to? Two dependent variables in this study, the actual fault corrections and refactoring changes reflect the stated purpose of the ASA tools. However, not all fault corrections or refactoring changes are equal, but a large number of corrections or changes over many versions should minimize the impact of the variability. The number of faults that are correctly identified and number of correctly predicted refactorings also reflect the purpose of the ASA tools. The number of false positives represents the increased effort required for a developer to determine whether a reported concern indicates a real fault or a design anomaly that should be refactored.

Content validity refers to the "representativeness or sampling adequacy of the content ... of a measuring instrument" [10]. In particular, the content validity of this research depends on whether the individual measures of actual fault corrections and refactoring changes adequately cover the notion of faults and needed refactorings. Surely there are faults that have not been corrected and code improvements that were not done. However, fault corrections and refactorings that were actually done represent the portion of faults and possible refactorings that developers deemed most important.

Internal validity focuses on cause and effect relationships. The notion of one thing leading to another is applicable here and causality is critical to internal validity. A statistical analysis was not needed due to the large number of false positives for fault concerns and refactoring concerns flagged by the ASA Tools, and the large number of false negatives reported for fault concerns. It is easy to show temporal precedence — evidence that cause precedes effect [4, 21], as we applied the ASA tools to software versions and looked at the fault corrections and refactorings that were performed to develop later versions.

At least one threat to internal validity remains. Fault corrections and refactorings may still be performed on versions that follow the ones included in the study. It is possible that, in the hypothetical alternative universe, after receiving the ASA tool reports, developers would have fixed some of the faults or performed refactorings on concerns that we classified as false positives. These concerns might be the same as those that are eventually fixed in the versions (in our real universe) that are released after those included in the study. In this situation, these concerns should not be counted as false positives. This threat can be lessened, but not eliminated, by further study of changes in subsequent versions.

External validity refers to how well the study results can be generalized outside the scope of the study [22]. The external validity of this study is limited mainly by three factors: the data sets, the ASA tools, and the sources of faults data. We studied only two OSS projects both implemented in one programming language (Java). There is no evidence that the results can be extended or generalized to other projects (even for other Java projects). To minimize the effect of this factor, we selected the projects from different domains, with varied number of developers and with variant sizes. In future work, we plan to study additional projects, implemented in other languages and with varied attributes in order to confirm our results.

The study was limited to evaluating the use of three ASA tools, and other tools may perform better or worse. The choice of particular ASA tools can affect our results; some tools may fail to find some coding concerns that are related to particular faults. However, the use of three different tools with different characteristics minimizes the effect of this threat.

6 Conclusions and Future Work

To evaluate the ASA tools, we examined the CVS repositories of two OSS projects in SourceForge [20]. We separated the fault fixes and refactorings from other updates and then ran the ASA tools to see if they report coding concerns that correspond to the actual changes. Our results are as follows:

- The fault detection ratios of the ASA tools were below 3%. Thus, the ASA tools would only minimally help the developers to detect the causes of future reported failures.
- Some ASA tools can help to produce well-documented software and identify unused data and code. For example, IDEA predicted approximately 71% of the refactorings that were performed.
- ASA tools report a large number of coding concerns that are not real faults or necessary refactorings. Developers using ASA tools must examine many false positives to decide which ones are real.
- The ASA tools vary greatly in their sensitivity. For example, IDEA reported approximately 44 times (4333/98) the number of refactorings that are reported by FindBugs for iText. In addition, Jlint failed to identify any refactorings or faults in either iText or jEdit.

In conclusion, our results indicate that the ASA tools that we evaluated are not effective in detecting the faults that are eventually reported. However, two of the tools can find program structures that are candidates for refactoring. The cost of using these tools may be prohibitive due to the large number of false positives.

Our research focused on the use of three ASA tools for Java programs. In future work, we plan to examine coding concerns reported by additional ASA tools, and study software written in other programming languages such as C/C++. Also, we plan to identify the types of faults that ASA tools can detect more successfully.

References

[1] artho.com. Jlint, Java program checker. http://artho.com/jlint, 2007. Last access: 12/25/2007.

[2] V.R. Basili and H.D. Rombach. The TAME project: Towards improvement-oriented software environments. *IEEE Trans. Software Engineering*, SE-14(6):758–773, June 1988.

[3] V.R. Basili and D. Weiss. A methodology for collecting valid software engineering data. *IEEE Trans. Software Engineering*, SE-10(6):728–738, 1984.

[4] D. Campbell and J. Stanley. *Experimental and Quasi-Experimental Designs for Research*. Houghton Mifflin Co., Boston, 1966.

[5] R. Chillarege, I. S. Bhandari, J. K. Chaar, M. J. Halliday, D. S. Moebus, B. K. Ray, and M. Y. Wang. On the value of static analysis for fault detection in software measurements. *IEEE Trans. Software Engineering*, 18(11):943–956, Nov 1992.

[6] K. Crowston, J. Howison, and H. Annabi. Information systems success in free and open source software development: Theory and measures. *Software Process Improvement and Practice*, 11:123–148, 2006.

[7] J. Friesen. Tools of the Trade, Part 1: Creating PDF documents with iText. *AdobePress*, 2005.

[8] Jedit.com. jEdit - Programmer's Text Editor. http://www.jedit.org/, 2007. Last access: 12/25/2007.

[9] JetBrains Co. IntelliJ® IDEA 7.0. http://www.jetbrains.com/idea/, 2007. Last access: 12/25/2007.

[10] F. Kerlinger. *Foundations of Behavioral Research, Third Edition*. Harcourt Brace Jovaonvich College Publishers, Orlando, Florida, 1986.

[11] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard P. W. Jones D. C. Hoaglin K. El Emam, and J. Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Trans. Software Engineering*, 28:721– 734, Aug 2002.

[12] S.C. Kothari, L. Bishop, J. Sauceda, and G. Daugherty. A Pattern-Based Framework for Software Anomaly Detection . *Software Quality Journal*, 12(2):99–120, 2004.

[13] B. Lowagie and O. Soares. iText, a free JAVA-PDF library. http://www.lowagie.com/iText/, 2007. Last access: 12/25/2007.

[14] W. Munger, J. Bieman, and R. Alexander. Coding concerns: do they matter? In *Workshop on Empirical Studies of Software Maintenance (WESS 2002)*, 2002.

[15] N. Nagappan and T.Ball. Static analysis tools as early indicators of pre-release defect density. In *Proc. 27th Int. Conf. Software engineering (ICSE'05)*, pages 580–586, 2005.

[16] J. Nunnally. *Psychometric Theory, Second Edition*. McGraw-Hill, New York, 1978.

[17] University of Maryland. FindBugs, Find Bugs in Java Programs. http://findbugs.sourceforge.net, 2007. Last access: 12/25/2007.

[18] Reasoning Inc. Automated software inspection: A new approach to increase software quality and productivity. Technical report, Reasoning, LLC, 2003. http://www.reasoning.com/pdf/ASI.pdf.

[19] N. Rutar, C. B. Almazan, and J. S. Foster. A comparison of bug finding tools for java. In *Proc. 15th Int. Symp. Software Reliability Engineering (ISSRE'04)*, pages 245–256, 2004.

[20] SourceForge Project. The Open Source software development web site. http://sourceforge.net/index.php, 2007. Last access: 12/25/2007.

[21] L. Votta and A. Porter. Experimental software engineering: A report on the state of the art. *Proc. 17th Int. Conf. Software Engineering (ICSE'95)*, 1995.

[22] C. Wohlin, P. Runeson, M. Host, M. Ohlsson, B. Regnell, and A. Wesslen. *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers, Norwell, MA, 2000.

[23] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. Hudepohl, and M. Vouk. On the value of static analysis for fault detection in software. *IEEE Trans. Software Engineering*, 32(4):240–253, April 2006.