An Approach to Merge Results of Multiple Static Analysis Tools

Na Meng, Qianxiang Wang, Qian Wu, Hong Mei
School of Electronics Engineering and Computer Science, Peking University
Key laboratory of High Confidence Software Technologies (Peking University)，Ministry of Education
Beijing, China, 100871
{mengna06, wqx, wuqian08, meih}@sei.pku.edu.cn

Abstract

Defects have been compromising quality of software and costing a lot to find and fix. Thus a number of effective tools have been built to automatically find defects by analyzing code statically. These tools apply various techniques and detect a wide range of defects, with a little overlap among defect libraries. Unfortunately, the advantages of tools' defect detection capacity are stubborn to combine, due to the unique style each tool follows when generating analysis reports. In this paper, we propose an approach to merge results from different tools and report them in a universal manner. Besides, two prioritizing policies are introduced to rank results so as to raise users' efficiency. Finally, the approach and prioritizing policies are implemented in an integrated tool by merging results from three independent analyzing tools. In this way, end users may comfortably benefit from more than one static analysis tool and thus improve software's quality.

1. Introduction

In recent years, some tools have been developed to automatically find defects in software, such as ESC/Java [1], Bandera [2], Proverif [3], PREfix [4] and FindBugs [5, 6]. Using different analyzing techniques, such as theorem proving [7], model checking [8], abstract interpretation [9], symbolic execution [10], syntactic pattern matching and data flow analysis [6], these tools usually produce different information about defects. The information, as mentioned in [11], collaboratively covers a wide range in the kinds of bugs found, with little overlap in particular warnings. Besides, the information also concerns about a large volume of warnings, which makes it difficult to know which to look at first.

Therefore, enlightened by [11], which mentioned that there is a need for a meta-tool to combine results of multiple tools together and cross-reference their output to prioritize warnings, in this paper, we attempt to explore an approach to merge results from different tools and bring forth a report with warnings uniform in style, so that users will benefit a lot from the fruit of different research teams working on developing static analysis tools.

Although the vision of merging analyzing result is very clear, there are still some problems we have to be faced with. For example, how to layout results from different tools as if they are from a single tool? How to prioritize results so that users will spend their effort and time effectively on important defects?

The main contributions of this paper are as follows:

- We propose an approach to merge results from several static analysis tools.
- We provide a general specification for each defect pattern mentioned in every tool so that they keep a consistent description style.
- We also propose two policies to prioritize results, so that users will be guided to decide which warning to check first.

The rest of the paper is organized as follows: we discuss the approach to merge results from tools in Section 2. Next, the general specification and prioritizing policies used in the approach are explained at length in Section 3. Afterwards, the merged result from the tool implementing our approach is shown in Section 4. Finally, Section 5 concludes the whole paper.

2. Approach overview

In this section, we first present an example to show the fact that different tools take care of various categories of defects with a little overlap among themselves; then expatiate on how to combine various tools' advantages.

1550-6002/08 $25.00 © 2008 IEEE
DOI 10.1109/QSIC.2008.30

2.1. An example

```
    ... ...
2  import java.util.zip.*;
3  import java.util.zip.ZipOutputStream;
4  public class Test {
5    static int BUFFER = 2048;
6    private String file = new String();
7    public synchronized void readFileName() throws Exception{
8      BufferedReader in = new BufferedReader(new FileReader("filename.txt"));
9      file = in.readLine();
10     makeZipFiles(file);   }
11   public synchronized void makeZipFiles(String file){
12     try { readFileName();
13         ZipOutputStream out =
                new ZipOutputStream(new BufferedOutputStream(new FileOutputStream("dest.zip")));
14         BufferedInputStream origin = new BufferedInputStream(new FileInputStream(file), BUFFER);

           ... ...
17         out.close();
18         origin.close();
19       } catch (Exception e) { e.printStackTrace();}
20   }}
```

Figure 1. Sample Java code

Our approach has been based on three tools so far: FindBugs [5], PMD [12] and Jlint [13]. The sample code partially shown in Figure 1 is compiled successfully by Java 1.5 compiler of Sun without error or warning. However, when brought to static analysis tools, it turns out to be defective.

When brought to FindBugs, the code is found to have 4 defects in all. The information includes: (1) line 6 invokes inefficient new String() constructor; (2) method readFileName() may fail to close stream created in line 8; (3) method makeZipFiles() may fail to close stream created in line 13 on exception; (4) method makeZipFiles() may fail to close stream created in line 14 on exception.

However, when we examine the code with PMD, 19 defects are listed. Among them, one reports the performance degradation in line 6 once more, one mentions "Avoid duplicate imports such as "java.zip.ZipOutputStream in line 3", and the others talk about better programming practice.

Then we use Jlint to check the same code. It only provides one warning saying that "Field 'BUFFER' of class 'Test' can be accessed from different threads and is not volatile" in line 14. Although such comment does not reveal enough hints; with more examination, we eventually deduce that synchronized methods readFileName() and makeZipFiles() invoke each other unconditionally circularly.

2.2. Our approach

With the results obtained from the three tools (FindBugs, PMD and Jlint), it is obvious that none of them is the "best" to subsume all useful defect information while holding no false positive at the same time. Consequently, there is a necessity to merge outputs so that benefits of each tool can be exploited adequately by users. Such exploitation includes taking into consideration all useful information from various tools, and paying attention to defects mentioned by more than one tool or emphasized by at least one tool. And Figure 2 illustrates our solution.

In Figure 2, there are two front ends to acquire inputs: one for checked programs and the other for defect pattern selection information. Particularly, to make it convenient for a user to choose patterns he or she is interested in, the user interface will display all defect patterns supported by the integrated tools in a uniform manner and organize these patterns according to categories (to be mentioned in Section 3) they belong to.

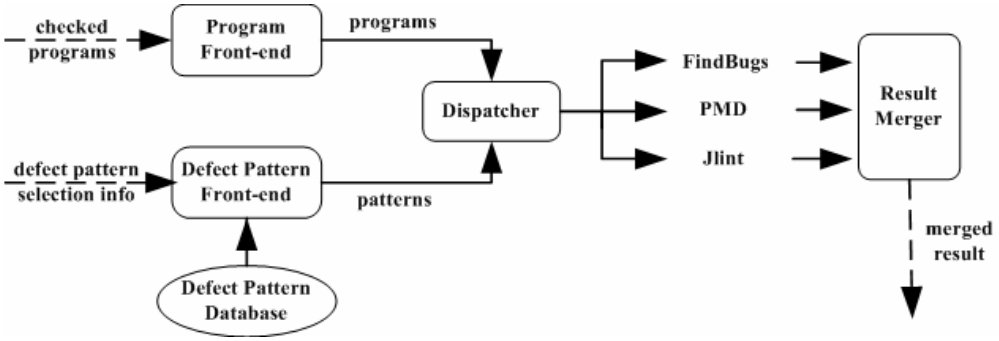

Figure 2. Architecture to merge tool results

Next, the input from users will be passed on to "Dispatcher". It is used to dispatch selected patterns and programs to one or more tools that are able to discover the defects. When giving out selected patterns, "Dispatcher" first makes a decision on which tool has the ability to discover a certain defect, then converts its general specification to the specific description in the tool in order that the tool can fulfill its task, and finally transmits the converted information to corresponding tool.

After getting information sent by "Dispatcher", different tools such as FindBugs, PMD and Jlint will perform their respective work and report defects found in the programs. All of these reports are sent to "Result Merger"—which is used to accomplish result combination. To achieve the purpose, "Result Merger" maps specific descriptions of patterns in a tool back to their general specification to keep a uniform reporting format, and applies prioritizing policies (to be expatiated on in Section 3) to attract more of users' attention to more important defects.

3. General specification and prioritizing policies used in the approach

As mentioned above, general specification helps customers a lot to choose defect patterns they concern, understand warnings from different tools with ease, and pay more attention to defects with higher priority—which is evaluated following certain prioritizing policies. Therefore, this section is organized to explain general specification first and then prioritizing policies.

3.1. General specification

The general specification for each defect pattern contains three main portions: a summary, which is recapitulated according to the pattern's description in one or more tools; the category it belongs to according to its appearance, and the category it belongs to according to the possible result which can be led to if it is not fixed. The general specification and tool specific description(s) for each defect pattern may be consulted by "Dispatcher" and "Result Merger".

3.1.1. Categories based on appearance

Figure 3 and Figure 4 illustrate the taxonomy on defect patterns according to their appearance, which is organized based on Java language's elements (for example, class, method and field). Conceptually, there are two major categories of defect patterns—patterns about defects independent of any library—named as "LIBRARY INDEPENDANT", as well as those specific to a certain library—named as "LIBRARY SPECIFIC".

3.1.2. Categories based on result

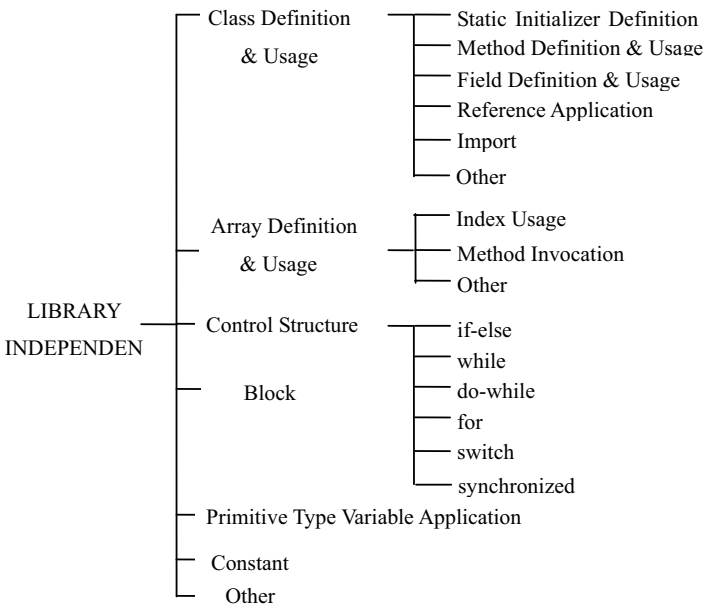

Figure 3. Taxonomy on library independent defect pattern

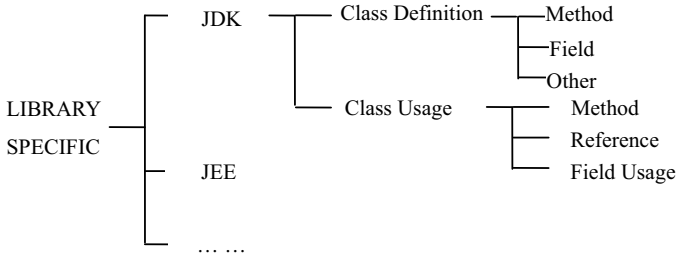

Figure 4. Taxonomy on library specific defect patterns

In addition to the categories on patterns based on their appearance, there are still other categories based on their possible results, such as "Error", "Fragile", "Security vulnerability", "Suspicious", "Performance degradation", "Dead code" and "Bad style". We will explain them one by one, and give an example when necessary.

"**Error**" category includes patterns of defects that will lead programs to abnormally exit, throw out exception, behave in a wrong way or violate some predefined rules or invariants.

"**Fragile**" category includes defects that will produce unexpected results under certain circumstance or prevent programs from reuse or extension. An example is synchronization on an updated field. Once the field is updated, wrong result may be generated.

"**Security vulnerability**" category includes defects that contain vulnerability for attacks of malicious code, such as SQL injection.

"**Suspicious**" category includes defects that generate some results contradicting common sense. For instance, when defining a class, a method with a return type is declared to hold the same name as its class.

"**Performance degradation**" category includes defects representing unnecessary computations, useless processes and inefficient ways of fulfilling tasks. For example, a variable is compared with itself just to produce "true" Boolean value.

"**Dead code**" category includes defects implying that some code is not executed at all or certain passed-in parameters are not utilized.

"**Bad style**" category includes defects telling

something about bad programming habits or elusive code, such as a local variable obscuring a field.

3.2. Prioritizing policies

When results produced by different tools have been converted to keep a consistent descriptive style, our approach applies two policies to rank them so that important and credible reports come before unnecessary and false reports.

The first policy is to rank reports according to their categories on result. The reports falling into “Error” class come first, while the ones falling into “Bad style” class come last.

The second policy is applied to reports falling into the same class. If a single defect is reported more than once, which means that more than one tool has found that defect and thus the integrated tool is more confident about the detection, the defect’s rank is raised and all relevant reports are integrated.

4. Experiment

In our experiment, we have implemented the architecture shown in Figure 2. And the report for the example discussed in Section 2 is partially displayed in Figure 5. The versions of FindBugs, PMD and Jlint integrated are 1.2.1, 4.1 and 3.0, respectively.

```
<?xml version="1.0" encoding="UTF-8"?>
<FindBugs_PMD_Jlint version = "1.2.1_4.1_3.0">
  <CheckedProgram>  D:\runtime-EclipseApplication\Experiment.zip  </CheckedProgram>
  <class name="Test">
    <defect beginline="8" endline="8" type="Error" category="ClassUsage_MethodInvocation" source="FindBugs">
      <summary>  InputStream: close() is not called./OutputStream: close() is not called.  </summary>
    </defect>
    <defect beginline="13" endline="13" type="Error" category="ClassUsage_MethodInvocation" source="FindBugs">
      <summary>  InputStream/OutputStream: close() may be not called on exception.  </summary>
    </defect>
    <defect beginline="14" endline="14" type="Error" category="ClassUsage_MethodInvocation" source="FindBugs">
      <summary>  InputStream/OutputStream: close() may be not called on exception.  </summary>
    </defect>
    ... ...
    <defect beginline="6" endline="6" type="Performance degradation"
            category="ClassUsage_MethodInvocation" source="FindBugs, PMD">
      <summary>  String: constructor is called with a parameter String.  </summary>
      <description>
        Avoid instantiating String objects; this is usually unnecessary.
      </description>
    </defect>
    ... ...
```

Figure 5. A merged report

The bugs detected by these three tools are named as “defects” uniformly. Each defect contains the following information, begin line and end line—to imply the context, type—to identify its category based on result, category—to identify its category based on appearance, source—to tell which tool reveals the defect, summary—to outline the defect, and occasionally description—to give more information. Except for “type”, “category” and “summary”, elements introduced above are extracted from information

supplied by different tools.

Besides, the defects are sorted using the following criteria sequentially: class concerned, type belonging to and degree of soundness. First of all, defects are sorted by classes they concern. In this way, users can focus on defects found in one class. In Figure 5, all defects listed between <class> and </class> labels reveal defects discovered in the Java class "Test". Second, defects referring to the same class are sorted by types and ranked by their categories based on result. Third, defects of the same type are sorted by degree of soundness. To reduce the negative influence of useless information (as mentioned in Section 2), we list the defects reported by more than one tool in front of those reported by only one tool. The idea comes from the intuition that for each defect, the more tools revealing its existence, the more confident our integrated tool becomes about the information.

5. Conclusion

In this paper, we compared and merged the analysis results from several static defect pattern based tools to exploit the characteristic that results from various tools collaboratively cover a wide range of defects, while holding a little overlap among them.

Actually, we have only merged results from three static analysis tools for Java to implement our approach. In future, we would like to take into consideration more tools about more programming languages. Besides, the categories discussed in Section 3.1 are only applicable to common defects. As our research goes on, we will improve the category hierarchy as much as possible.

This paper is supported by the National High-Tech Research and Development Plan of China, No. 2006AA01Z175, and National Natural Science Foundation of China, No. 60773160.

References

[1] K. Rustan, M. Leino, G. Nelson, and J.B. Saxe. Esc/Java user's manual. Technical note 2000-002, Compaq Systems Research Center, October 2001.

[2] J. Corbett et al. Bandera: Extracting finite-state models from Java source code. In Proc. 22nd ICSE, June 2000.

[3] Goubaut-Larrecq J, Parrennes F. Cryptographic protocol analysis on real C code. In: Cousot R, ed. Proc. of the 6th Int'l Conf. on Verification, Model Checking and Abstract Interpretation. LNCS 3385, Paris: Springer-Verlag, 2005. 363−379.

[4] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A Static Analyzer for Finding Dynamic Programming Errors. Software – Practice and Experience (SPE), 30: 775 - 802, 2000.

[5] FindBugs, http://findbugs.sourceforge.net.

[6] D. Hovemeyer and W. Pugh. Finding Bugs is Easy. In Proceedings of the Onward! Track of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), 2004.

[7] D. L. Detlefs, G. Nelson, and J. B. Saxe. A theorem prover for program checking. Technical Report, HP Laboratories Palo Alto, 2003.

[8] E.M. Clarke, Jr. O. Grumberg, and D. A. Peled. Model Checking. MIT Press, 2000.

[9] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Proc. 4th POPL, pages 238–252. ACM, 1977.

[10] Boyer RS, Elspas B and Levitt KN. SELECT—a formal system for testing and debugging programs by symbolic execution. Proceedings of the International Conference on Reliable Software, Los Angeles, CA, 21–23 April 1975; 234–245.

[11] Nick Rutar, Christian B. Almazan and Jeffrey S. Foster. A Comparison of Bug Finding Tools for Java. Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE'04).

[12] PMD/Java, http://pmd.sourceforge.net.

[13] Jlint, http://artho.com/jlint.

[14] Checkstyle, http://checkstyle.sourceforge.net.