Towards SOA-based Code Defect Analysis

Qianxiang Wang, Na Meng, Zhiyi Zhou, Jinhui Li, 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
\{wqx, mengna06, zhouzhy06, lijh06, meih\}@sei.pku.edu.cn

Abstract

During recent years, a number of defect pattern based static code analysis tools have become more and more admiring since they are able to help discover vital defects. Coinciding with the trend is people’s increasing demand for availability as well as facility of various powerful static code analysis techniques. Currently, these techniques are usually released as independent installable programs or plugins for integrate development environment (IDE). However, in this paper, we introduce a new approach—the Service Oriented Architecture (SOA) based approach—by publishing defect checking capabilities as a central service with standard web service interface provided in order to facilitate customers to take advantage of benefits coming along with SOA. This approach can serve as a complement to existent ways since its usage and service maintenance are much simpler and related data collection will become more effective.

1. Introduction

Static code analysis is the analysis of computer software that is performed to acquire information concerning the dynamic behavior of programs built from that software even without actually executing the programs. It can be used in a variety of software engineering tasks, such as architecture recovery, assertion discovery, debugging, automotive software engineering, clone detection, program comprehension and reverse engineering [1].

This paper focuses on how to use code analysis to find code defect. There are several ways for classifying defects, such as Orthogonal Defect Classification [2]. This paper divides defects into three levels, from view of the defect checking rules: 1) Defects in the first level can be found by comparing the code with language grammar, which is usually specified using BNF. This kind of defect can be found easily by Compiler; 2) Defects in the second level can be found by comparing the code with various defect patterns [3, 4, 5], which are gained from experience, and application semantic independent; 3) Defects in the third level can be only found by comparing the code function with application requirement. These defects are application specific, and are usually found by software testing, which is labor intensive.

Some of the more well-known static tools include Coverity [3], FindBugs [7], JLint [11] and PMD [8], all of which are based on defect patterns—code idioms that are often errors. And the fact that people tend to fall into the same traps by committing similar errors repeatedly supplies these tools with fantastic opportunities to survive and even become quite popular.

1.1 Challenges

Accompanying the growing popularity of defect pattern based static code analysis tools is people’s rising requirement for availability as well as facility of these
tools.

As is exemplified in [6], there is no single “best” defect-finding tool to discover all real defects contained in checked programs but not provide any false report. Therefore, in order to benefit from advantages of different analysis tools when searching for as many useful defects as possible, usually people need to download one tool after another to their own computers and then install them patiently since most of these tools are currently issued as independent software or plugins for IDE.

Additionally, what intensifies people’s demand for convenience as well as simplicity of the tools is the fast growth in the number of defect patterns. Owing to the spreading use of various middleware by companies and organizations, defect patterns coming along with intermediate software are puzzling more and more programmers. Consequently, users always have to update their tools so as to keep abreast with the present state of defect pattern libraries, much like the way they deal with the Antivirus software.

Confronted with people’s urgent want for convenience as well as easiness of code defect analysis techniques, we propose an approach to meet the need.

1.2 Approach overview

In our approach, we integrate defect checking techniques into a server and publish its code defect analysis capability as Web Service, named as Code Defect Analysis Service (CODAS), which is based on SOA and focuses on the matching of multifarious defect patterns.

Compared with practical defect pattern based static code analysis tools, this approach integrates vantages of different analysis technologies employed by various tools and thus releases people from troubles concerning downloading and installing of different software. Besides, the network-enabled service is undoubtedly handy and simple to utilize for customers. Furthermore, the published service may facilitate collection of analysis results and feedback from users, both of which may avail to improve quality of service provided.

In addition, different from many current SOA applications dealing with how to conform to business processes, this application focuses on how to produce high-quality applications complying with diverse business logic. By submitting source code or compiled bytecode to the service, developers and testers may get to know more defects and corresponding possible aftereffects. In this way, programmers tend to fix more bugs earlier with less cost.

1.3 Paper organization

The rest of the paper is organized as follows. In section 2, we introduce some important static code analysis tools, which enlighten our research work. Section 3 describes design of CODAS. Section 4 gives some experiments with a prototype of CODAS to demonstrate the feasibility of our approach. Section 5 discusses related issues and future work. Finally, Section 6 concludes the whole paper.

2. Related work

The motivation of our approach is sparked by several static code analysis tools. The most significant three ones among them include JLint [11], FindBugs [7], and PMD [8].

**JLint** analyzes Java bytecode with syntactic checks and dataflow analysis. What distinguishes the tool from other Java code defect checkers is its interprocedural and inter-file component, which finds deadlocks by building a lock graph and ensuring that there are never any cycles in the graph. Up till now, this tool has only been published as independent installable programs for different Operating Systems.

**FindBugs** also analyzes Java bytecode as JLint does, but applies a series of ad-hoc techniques designed to balance precision, efficiency and usability. The distinct predominance of this tool is that it contains a considerable number of defect patterns not covered by any other tool. Moreover, it is released as both
independent software and Eclipse plugin for customers with different appetites. Especially, with a 1.4 or better Java Runtime Environment (JRE) installed, users are sponsored to try the graphic user interface (GUI) version of FindBugs on their projects using Java Web Start. Such funny trial renders us inspiration for the approach proposed in this paper. Besides, the proprietary solution composed with FindBugs—Fortify [9] once encouraged users to submit code to its web site, and gave back analysis results to them several days later. This attempt also contributes to our research.

PMD, unlike FindBugs and JLint, performs defect checking on Java source code. In such a manner, the tool can discover some suspicious or abnormal coding practice which implies serious defects but should have been dropped by analysis tools dealing with Java bytecode due to the optimization conducted by compilers. Additionally, the tool has already been published as independent software as well as plugins for various IDEs.

With the summary of tools aforementioned, we notice again that various tools are preeminent in dealing with different kinds of defect pattern matching. Naturally, an integrated tool converging benefits of them all will certainly free customers from disturbance of installing analysis tools one by one.

Therefore, we will encapsulate the checking capabilities of various tools into a Web Service instance. This manner can not only guarantee that users always have access to its latest version—just as Java Web Start does, but also ensures that its developers are able to have enough analysis results to evaluate and thus decide how to improve the service provided—simply like what Fortify does.

Based on the consideration remarked above, we bring forward our SOA-based code defect analysis approach, which is to be expatiated on in the following section.

3. Code Defect Analysis Service (CODAS)

In this section, we first illustrate the architecture of CODAS, then introduce interface of the service, and finally discuss the service maintenance.

3.1 Architecture of CODAS

As is shown in Figure 1, CODAS’ architecture consists of the following major parts:

User Preference Customization is exploited by users to customize some information according to their own preference. The information may involve what kind of defects users would like the service to concern and in what manner they expect the service to present discovered defects. For instance, with the help of Defect Examination module, users may deselect defect patterns about concurrency if they are aware that the checked code has nothing to do with multithread programming. In this way, the service is configured to focus on only defect patterns customers may be interested in and thus work out results sooner. On the other hand, by virtue of Defect Presentation module, users can also decide whether defects should be organized according to their seriousness or the checked code they enclose, as well as whether defect reports should be presented directly or stored into a designated file.
such as source code and object code.

**Analysis Engine Factory** contains a number of core analysis engines which implement different analysis techniques applied to various code forms of diverse languages aiming at a wide range of defects. The engines in the factory form up a hierarchical structure because of information dependence. For instance, the call graph analysis in Analysis Engine 2 depends on info obtained from control flow graph analysis and hierarchy graph analysis in Analysis Engine 1.

**Detector Category Collection** includes a system of detector categories, which is formed up in accordance with mechanisms used to detect bugs. For instance, detectors for dead store, uninitialized variable and useless variable belong to the same category since they all infer conclusions from intraprocedural data flow analysis results.

**Pattern Set Repository** comprises diverse defect pattern sets. The partition of these pattern sets relies on the detectors applied. In other words, if two defect patterns can be discovered with the same analysis logic, they may belong to the same defect pattern set. Additionally, for more information about defect patterns as well as their category hierarchy, please refer to [12].

**Sub Task** is a concept introduced to reduce time spent on program analysis. As detectors in the same category usually share related information, while detectors from distinct categories have less in common; we may divide the overall task of defect analysis into several sub tasks to increase parallelism and improve response time of the service.

### 3.2 Service interface

As a kind of Web Service, CODAS owns interface definition to facilitate customers when they are submitting code. The definition involves the following key issues.

- **Transport Protocol**
  
  The protocol responsible for transporting messages between the service and its consumers is SOAP.

- **Code Form**
  
  The code to upload can be source code or object code. Certainly, the size of object code is usually smaller than that of its corresponding source code and thus requires less bandwidth. However, as mentioned in Section 2, the upload of source code is not always undesirable since it may supply more hints of defects and thus do customers an important favor.

- **Auxiliary Information**
  
  Some additional information which may assist the server to better carry out code defect analysis can be mention of programming language used in the inspected code, compiler’s name or version if the code has already been compiled, or middleware used if the code is applications based on intermediate software.

### 3.3 Service maintenance

With the continuous emergence of new software, especially with the endless appearance of novel software framework and middleware, a growing number of defect patterns come forth and are playing an increasingly momentous role. A rising number of programmers are trapped by defects when developing their own software on the basis of framework and middleware. Consequently, it is a crucial task to frequently update the Pattern Set Category displayed in Section 3.1 so that it may cover more defect patterns.

In addition, with an eye to the development of static code analysis, we also strive to endow extensibility of Analysis Engine Factory discussed in Section 3.1 in order that new analysis engines may be appended when necessary.

### 4. Experiments

To verify the feasibility of our approach, we have implemented a prototype of CODAS on our web site http://as.pku.edu.cn. Although its analysis capability is not as powerful as anticipated, the cheerful results of
experiments performed on it fortify our confidence in the approach.

4.1 Design of experiments
We design two sets of experiments to substantiate the effectiveness of our approach through comparison.

In the first set of experiments, we analyze code defects existent in Apache Tomcat 5.5.26 with FindBugs and PMD; both of which are installed on a client computer and function to examine for bugs locally. Meanwhile, in the second set of experiments, we check the same code with CODAS. Encapsulating the capabilities of FindBugs and PMD, CODAS is provided by a server for users who favor remote code analysis.

By recording and comparing the time spent by each tool, we can deduce whether the cost CODAS pays for its availability and facility is worthwhile. Specially, so as to elude inaccuracy caused by accidental factors, we conduct analysis with each tool for ten times and then take the average time span as a final result shown in Table 1.

By the way, the server side application CODAS is deployed on a personal computer (PC) with a Pentium 4 2.26G CPU, a 1G memory as well as Windows XP Professional Edition SP2 operating system. And the network used is 100MB local area network (LAN).

4.2 Results
Running time of the two analyzing modes—local mode with FindBugs as well as PMD and remote mode with CODAS—is displayed in Table 1.

In this table, we recorded the time spent by FindBugs and PMD separately when they check defects locally. On the other hand, we also time CODAS when the same code is examined remotely.

As is shown in Table 1, the total running time used for comparison consists of time taken for uploading as well as analyzing checked code, and downloading defect reports. In local analysis mode, however, since there is no need to upload or download data from a remote server; the time spent on these two tasks is zero.

<table>
<thead>
<tr>
<th>Analysis Mode</th>
<th>Up-</th>
<th>Analyze</th>
<th>Down-</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local</td>
<td>Bin</td>
<td>0</td>
<td>42047</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Src</td>
<td>0</td>
<td>1030781</td>
<td>0</td>
</tr>
<tr>
<td>Remote</td>
<td>Bin</td>
<td>3781</td>
<td>42844</td>
<td>422</td>
</tr>
<tr>
<td></td>
<td>Src</td>
<td>3891</td>
<td>1039459</td>
<td>4234</td>
</tr>
</tbody>
</table>

In our experiments, we get 408 defects from FindBugs and 72286 from PMD. Correspondingly, CODAS provides the same number of defects as FindBugs does when accepting bytecode and the same as PMD when getting source code.

Under the circumstances of maintaining checking capabilities owned by encapsulated techniques, CODAS usually requires more running time to function well. This fact is sound in each respect. For one thing, as a sort of service provided by a central server, CODAS needs time to accept uploaded checked programs from users and transport analysis reports as feedback. For the other thing, SOAP Message Transmission Optimization Mechanism (MTOP) encoding and decoding account for the additional cost occurring in analyzing time of remote mode.

What is more important, the extra time cost of CODAS is no more than twelve percent of the overall interval necessary for analysis. Therefore, the additional expense is acceptable and thus our approach is feasible.

5. Discussion and future work

Despite of convenience and simplicity of the service, CODAS can never totally substitute analysis tools published in other forms, such as independent software or plugins for IDE. On the one hand, the service has significant dependence on transportation conditions of network. On the other hand, as multiple users submit code simultaneously, the server may have too heavy load to preserve satisfactory performance. And as our research work goes on, we would like to look for a better solution
that combines advantages of traditional manners and our approach.

Additionally, as is well known to all, no static code analysis is powerful enough to simulate the exact execution of any program or thoroughly excel testing when looking for defects. Therefore, in future, we will also endeavor to improve the combination between static analysis and dynamic execution information retrieval (such as online monitoring which is discussed a lot in [13]) so as to improve accuracy of analysis and reliability of final results.

Furthermore, the philosophy of developing software by the aid of SOA can be even extended to other phases of the whole lifecycle of software, such as requirement analysis, design, testing and maintenance. In this way, we may import a brand new conception of “SOA based software engineering”.

6. Conclusion

In this paper, we have introduced a new approach—SOA based approach—by encapsulating various analysis techniques into a kind of service to facilitate customers’ adoption of static code analysis when searching for defects in their own code. We have discussed about the design of CODAS, which embodies the central philosophy of our approach; we have also expounded the experiments carried out on a prototype implementing CODAS’ design to verify the feasibility.

Additionally, in future, we will perfect the implementation of CODAS by adding new analysis engines as well as defect patterns, reducing its dependence on performance of network as well as servers, and combining its static analysis capabilities with retrieval of programs’ dynamic execution information.

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