MPAnalyzer: A Tool for Finding Unintended Inconsistencies in Program Source Code

Yoshiki Higo
Osaka University
1-5 Yamadaoka, Suita, Osaka, Japan
higo@ist.osaka-u.ac.jp

Shinji Kusumoto
Osaka University
1-5 Yamadaoka, Suita, Osaka, Japan
kusumoto@ist.osaka-u.ac.jp

ABSTRACT

Unintended inconsistencies are caused by missing a modification task that requires code changes on multiple locations in program source code. In order to identify such inconsistencies efficiently, we proposed a new technique. It firstly learns how code fragments were changed in the past modification tasks, and then, it identifies where inconsistencies exist at the latest version. In this paper, we focus on an aspect of the tool that we developed and shows a case study that we conducted with the tool. A video of the tool is available at http://youtu.be/a7_FVVz4-v0.

Categories and Subject Descriptors

D.2.2 [Design Tools and Techniques]: Computer-aided software engineering; D.2.7 [Distribution, Maintenance, and Enhancement]: Version control

Keywords

Inconsistency detection; Modification patterns; Static analysis

1. INTRODUCTION

A modification task such as a bug fix occasionally requires source code changes on multiple locations. If a developer overlooks a location to be changed in a given task, an unintended inconsistency occurs there. Such an inconsistency may cause faults in the future, and such a task inherently involves the risk of overlooking locations [5].

Figure 1 shows two source files of Apache Subversion project. The 721th line of update_editor.c, the 1,340th and 1,502th lines of status.c include the same expression. Two of them were changed in the same way in 2004/01/10. Then, in 2004/03/11, the remaining one was modified and the 3 expressions got consistent again. The commit log of 2004/03/11 described that the modification was for following up to the modifications in 2004/01/10. The commit log is an evidence that: the 3 expressions must have been modified simultaneously; however the developer overlooked one of them.

Figure 1: Modifications in Subversion

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Copyright 2014 ACM 978-1-4503-3013-8/14/09 ...$15.00.
http://dx.doi.org/10.1145/2642937.2648616.
3 CDTs, but they have not been applied to detecting statement-level ones yet.

- Token-level inconsistencies can be detected by CDTs only if they are included in detected clones. If code surrounding inconsistencies is not duplicated to its correspondents, they will not be detected by CDTs. We are able to detect such clones like the 3 expressions in Figure 1 if we decrease the minimum size of clones to be detected. However, we will get a long list of clones even if the system is not large. Extracting necessary clones from it is a complicated and burdensome task.

As an alternative way to detect inconsistencies, we have proposed a new technique [1]. It firstly finds modification patterns, each of which means how a code fragment was changed to another one, by analyzing a code repository of target software. Then, it detects inconsistencies in the latest source code by using the patterns. In literature [1], we have reported the following experimental results.

- It detected 16 and 69 unintended inconsistencies from the two open source systems, respectively.
- Most of the detected unintended inconsistencies were not detected by two CDTs, CCFinder[3] and Nicad[6]. CCFinder detected 2 and 39 out of the 16 and 69 inconsistencies, and Nicad detected 2 and 2 out of them.

In this paper, we focus on an aspect of the tool we developed. The remainder of this paper is organized as follows: Section 2 explains our solution for the issues of inconsistency detection. Section 3 introduces our tool that we developed; Section 4 shows a case study we have conducted with the tool; Section 6 concludes this paper.

2. OUR SOLUTION

In order to detect where inconsistencies exist, we have proposed a new technique that utilizes the past changes [1]. The followings are its characteristics, and it is free from the issues of grep and CDTs.

- It can detect not only token-level inconsistencies but also statement-level ones.
- It can detect inconsistencies even if they are not in duplicated code chunk.
- It does not require deep analysis of source code, and so it is easy to apply it to many programming languages.
- It has high scalability, less than an hour is required for analyzing million lines of code and its history.

The technique analyzes how source code was modified in the past to find modification patterns. A **modification pattern** (in short, **MP**) means a pattern that a code fragment was changed to another one. For example, by analyzing modifications performed in 2004/01/10 in Figure 1, the technique finds that the following MP occurred twice.

\[
\text{eb->target} \rightarrow \#eb->target
\]

Found MPs are used for detecting inconsistencies. In the case of Figure 1, by using the above MP, we can automatically suggest that the 1,502th line must be changed to “#eb->target” just after the 2004/01/10 modifications.

3. TOOL: MPANALYZER

We have implemented the proposed technique as a tool, MPAnalyzer\(^3\). Currently, the tool handles C/C++ and Java software systems managed with Subversion.

MPAnalyzer has two functions called **mining function** and **detection function**: **mining function** is a batch processing that analyzes past modifications to find MPs; **detection function** is an interactive processing for detecting inconsistencies to be modified with the found MPs.

3.1 Mining Function

The input and output of **mining function** are as follows.

**Input:** a code repository of a target system,

**Output:** a set of MPs with **confidence** and **support** values.

**mining function** consists of three steps.

**STEP1:** identifies revisions where the source code was changed.

**STEP2:** extracts MPs from the revisions.

**STEP3:** calculates **confidence** and **support** metrics for MPs.

In **STEP1**, **mining function** identifies revisions where one or more source files were modified. Code repositories contain not only source files but also other kinds of files such as manual or copyright files. There are revisions that no source files were modified. Consequently, the purpose of **STEP1** is eliminating revisions to be ignored.

In **STEP2**, **mining function** extracts MPs from every of two consecutive revisions. MPs are extracted at the level of program statements and conditional predicates. Interface E in Figure 2(a) shows an extracted MP. The detailed operations in **STEP2** is described in literature [1].

In **STEP3**, **mining function** calculates **confidence** and **support** metrics for each of the extracted MPs.

**Definition 1 (Confidence)** This is a probability that a given code fragment \( (cf_1) \) is changed to another one \( (cf_2) \). **Confidence** is represented by a fraction, \( \frac{m}{n} \). Herein, “\( m \)” is the number of modifications whose pre-changed code fragments are \( cf_1 \) and post-changed ones are \( cf_2 \) \( (cf_1 \rightarrow cf_2) \). “\( n \)” is the number of modifications whose pre-changed code fragments are \( cf_1 \) \( (cf_1 \rightarrow *) \). \( m \leq n \) is always satisfied.

**Definition 2 (Support)** This is a numerical metric to present the number of equivalent modifications classified into a given MP. Two modifications, \( cf_3 \rightarrow cf_4 \) and \( cf_5 \rightarrow cf_6 \) are equivalent if the two conditions are satisfied.

- The token sequence of \( cf_3 \) is identical to the one of \( cf_5 \).
- The token sequence of \( cf_4 \) is identical to the one of \( cf_6 \).

3.2 Detection Function

The inputs of **detection function** are the followings, (1) source files where inconsistencies are to be detected, (2) a list of MPs with their **confidence** and **support** values, and (3) thresholds of **confidence**, **support**, and **place**. The output is a list of detected inconsistencies.

**detection function** consists of the following steps.

\(^3\)https://github.com/YoshikiNigo/MPAnalyzer

\(^3\)A type-3 clone is a duplicate code region including some instructions that are not duplicated to its correspondents.
**STEP1:** selects MPs.

**STEP2:** detects inconsistencies.

In **STEP1**, detection function selects MPs used for detecting inconsistencies. Selections are conducted by specifying the thresholds of confidence and support. If both the confidence and support values of a given MP are within the threshold range, it is used for detecting inconsistencies.

Figure 2(a) shows the window for selecting MPs. In the window, developers select MPs with interfaces A and B. The right-hand list (C) shows a selection result. Developers can see the commit logs where the MPs occurred and their code with interfaces D and E.

In **STEP2**, every of the source files is checked to find out if it contains any of the selected MPs. Assuming that there are a source file $f$ in a revision $r_1$ and a selected MP $mp = cf_1 \rightarrow cf_2$. If the token sequence of $f$ contains the token sequence of $cf_1$, MPAnalyzer regards that $f$ has an inconsistency, and it shows the following information:

- location of the matched code fragments (file path, start line, and end line), and
- $cf_2$, which is a suggestion of how the code fragments should be modified.

In detection function, we introduce one more metric, place. place represents the number of code fragments matched with $cf_1$. If $cf_1$ is matched in many locations, the matched locations are unlikely to be unintended inconsistencies. The authors think that the number of unintended ones is a small number because they were overlooked. Consequently, we use place to specify an upper limit of matched locations. If a code fragment is matched more often than the place threshold, the matched locations are not suggested to developers.

Figure 2(b) shows the window for detecting inconsistencies. In the window, developers firstly select a revision where inconsistencies are to be detected and they specify a place threshold in interface F. Then, they start a detection with the specified conditions by pushing the search button. Detecting inconsistencies usually finishes in less than 10 seconds, but of course it depends on the size of search targets. The detected inconsistencies are listed in interface G. Developers see their source code and suggestions of how to modify by using interfaces H and I. The commit logs of the matched MPs are available interface J.
4. CASE STUDY

We have applied MPAnalyzer to Apache HTTP Server\(^3\). In this case study, we used 4 branches, 1.3.x, 2.0.x, 2.2.x, and 2.4.x in the repository. Those branches are intended for providing stable versions of the software, and source code modifications in the branches are for bug fixes and refactorings rather than adding new functionalities.

The target period is from each start of the branches to the end of October, 2013. Table 1 shows the number of commits\(^5\) in the branches and size of the latest versions.

Table 1 also shows the quantitative result. The column names mean the followings.

- **Total MPs**: the number of all the found MPs.
- **Used MPs**: the number of the MPs used for detecting inconsistencies. In this case study, we selected MPs satisfying two conditions: (1) its support was 2 or more; and, (2) its confidence was 1.
- **Inconsistencies**: the number of inconsistencies detected by using the selected MPs. In this case study, we configured the threshold of place as 1.
- **UIs**: the number of unintended inconsistencies included in the detected inconsistencies. Herein, an unintended inconsistency means that it is a code fragment that was overlooked in a past modification task. The authors carefully browsed each of the detected inconsistencies to judge whether it was unintended or not.

<table>
<thead>
<tr>
<th>branch</th>
<th># of commits</th>
<th># of .c files</th>
<th>LOC</th>
<th># of total MPs</th>
<th># of used MPs</th>
<th># of inconsistencies</th>
<th># of UIs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.3.x</td>
<td>3,434</td>
<td>161</td>
<td>104,672</td>
<td>5,232</td>
<td>973</td>
<td>58</td>
<td>42</td>
</tr>
<tr>
<td>2.0.x</td>
<td>217</td>
<td>195</td>
<td>154,894</td>
<td>258</td>
<td>20</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2.2.x</td>
<td>831</td>
<td>225</td>
<td>193,011</td>
<td>1,240</td>
<td>122</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>2.4.x</td>
<td>510</td>
<td>250</td>
<td>210,217</td>
<td>1,090</td>
<td>69</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

We found 52 unintended inconsistencies in total. In our investigation, 19 out of them were classified into bug fixes, 17 were refactorings, 14 were functional enhancements, and the remaining 2 were for removing warnings in its compilation time. We spent about 5 hours to judge the 71 detected inconsistencies. But, if developers of the software had judged them, they would have spent much less time because they have deep knowledge on the software.

5. EXPERIENCES WITH MPANALYZER

Through this case study, we learned that the commit logs play an important role in identifying whether detected inconsistencies are unintended or not. It would be much more difficult to do it if we could not have seen the commit logs. Interface J of Figure 2(b) was very useful in the case study.

We found some trivial inconsistencies in addition to severe ones that may cause faults in the future. For example, the trivial ones include (1) missing modifiers such as const and (2) recommending renaming variables. If developers use MPAnalyzer within strictly limited time frame, such trivial inconsistencies will be obstacles to identify severe inconsistencies efficiently. Consequently, we need to invent a technique that ranks detected inconsistencies based on their severities.

MPAnalyzer is intended not for avoiding inconsistency occurrences but for detecting inconsistencies that have already occurred. In order to avoid occurrences, we need another tool. A plugin for IDEs seems to be a reasonable choice as an implementation of such a tool. By integrating the tool to IDEs, it can check whether developers have overlooked some locations to be changed in a given task before committing their changes. This case study showed that the MP-based approach is useful to detect inconsistencies, which means the approach is also promising to avoid inconsistencies.

6. CONCLUSIONS

In this paper, we introduced our tool, MPAnalyzer, which was developed for detecting unintended inconsistencies in program source code. MPAnalyzer takes a repository of target software to find patterns that code fragments were changed to other ones in the past modification tasks. After finding patterns, MPAnalyzer detects inconsistencies by using them. The pre-changed code fragments in the patterns are keys for searching inconsistencies. The post-changed code fragments in the matched patterns are showing how the detected inconsistencies should be modified.

We applied MPAnalyzer to four branches in the repository of Apache HTTP Server. As a result, we found 52 unintended inconsistencies. The application result shows that the pattern-based approach\(^1\) is promising and the tool is useful for efficient analysis of unintended inconsistencies.

7. ACKNOWLEDGMENTS

This work was supported by JSPS KAKENHI Grant Numbers 25220003, 24650011, and 24680002.

8. REFERENCES


\(^3\)http://httpd.apache.org/

\(^5\)Herein, this number means the number of commits where at least one .c file was modified in the branch.