A Preliminary Analysis and Case Study of Feature-based Software Customization (Extended Abstract)

Yufei Jiang, Can Zhang, Dinghao Wu, and Peng Liu
The Pennsylvania State University
University Park, PA 16802, USA
{*yzj107,czhang,dwu,pliu*}*@ist.psu.edu*

***Abstract*—Modern software engineering practice heavily relies on third party libraries, existing frameworks, high level programming languages, and agile development methodologies, which allows us build more complex software and deliver it faster. However, on the other hand, such practice causes some negative consequences such as bloatware and feature creep. A phenomenon of importing and implementing redundant features into software products has been observed in many software evolution and iteration lifecycles. In this paper, we describe an approach to customizing Java bytecode by applying static dataflow analysis and enhanced programming slicing technique. This approach allows developers to customize Java programs based on various users' requirements or remove unnecessary features from tangled code in legacy projects. Our preliminary evaluation and case study results show that our approach has the potential for practical use.**

***Keywords*-program analysis, bloatware, feature creep, software customization**

I. Introduction

A typical modern software system delivers a set of features to its users in a bundled way. The requirements of removing and customizing one or some of those bundled features are raised from both developers and users, for both software engineering reasons and software security reasons.

From software engineering perspective, with the rapid development of modern software engineering, the complexity of modern software keeps increasing, which causes the problem known as bloatware [1]. A type of bloatware is caused by the feature creep phenomenon. Apparently, feature-creep bloatware causes many negative consequences for the resulted software, including larger size, higher code complexity, and potential reliability and security issues. It is more difficult for developers to change, maintain, and manage the code. It also means longer testing time, more bug reports, and releasing patches in a higher frequency. From software security perspective, removing unnecessary features according to different requirements not only reduces the attack surface, but also achieves moving target defense by increasing software diversity.

Existing technologies and previous research on bloatware touch this problem from different perspectives and cannot solve this problem very well [2], [3]. Code review can help mitigate these issues to some degree in some scenarios. However, in many cases code review is not a feasible solution due to the cost or the availability of source code. The effort of automatically removing bloat from bloatware has been made by some researchers. However, they did not touch the bloatware that caused by feature creep. Some studies focus on code local optimization [2], [3], [4]. In this paper, we propose a novel approach to conducting feature-based program customization via multiple-step static analysis. One of the steps of our approach is based on an enhanced program slicing method called *solo slicing*. Based on a set of seed methods defining a feature, our approach investigates its call sites, return values, and parameters. Then starting from the return values and parameters at the call sites, we remove any code that depends on return values, and any code on which *only* the parameters depend. Next, we remove call site itself. At last, if possible, we remove method definition by checking a set of rules.

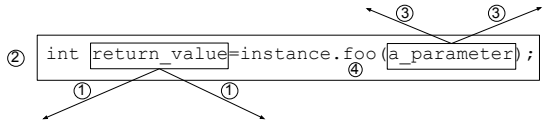

Figure 1: Delete Overview

II. Approach

Intuitively, a software feature can be represented by a set of data structures and methods that implement this feature. Many features cross cut with other business logic. So feature customization cannot be done by modifying one or several methods' definition. Features are usually implemented as many spread and repeated method calls. Therefore, we define the methods of interests as seed methods and define a *feature* as all call sites of seed methods.

Based on the definition, to remove a feature, we can clearly define our task is to remove all call sites of the seed methods *safely* and clear all the redundancy caused by this removal. Fig. 1 shows a mock call site. The arrows in the figure indicate potential dependency relationship with the context. The numbers in the figure denotes the index of the steps. To remove all methods invocation in a program, we potentially need to remove 4 parts of code in order. The first part is the code that depends on the return value or side effects (objects and array references that redefined in the callee) of the call site. The second part is the call site itself. The third part is the code on which *only* the parameters of the call site depend. And the last part is the method definition.

We perform feature-based customization in four steps:

Step 1: Forward Slicing: To remove the code that depends on the return value or side effects of the call sites, we take advantage of forward slicing technology.

Algorithm 1 Solo-slicing Algorithm

```
1: function SOLOSLICING(G,S)
2:     PDG ← G
3:     CallSiteVerticesSet ← S
4:     WorkList ← CallSiteVerticesSet
5:     while WorkList ≠ ∅ do
6:         Vertex ← WorkList.getOneVertex()
7:         PDG.removeGivenVertex(Vertex)
8:         PDG.updateVerticesOutDegree()
9:         for each vertex v changes its out degree do
10:            if v.OutDegree = 0 then
11:                WorkList.add(v)
12:            end if
13:        end for
14:    end while
15: end function
```

Specifically, we use return values and side effects of the call site as slicing criteria. The forward slicing algorithm searches the System Dependency Graph (SDG) to slice out the statements that depend on the slicing criteria via data flow or control flow. By removing the sliced out statements, we make sure that the program is still runnable after seed method call sites removal.

Step 2: Call Sites Delete: In this step, we remove all call sites of interests. In this step, we successfully disable a feature of the program. For example, by removing all call sites of method java.io.DataInputStream.read(), we customize an original program into a data-reading-free program.

Step 3: Solo Slicing: There are some statements on which *only* the actual parameters of seed method call sites depend. After call sites removal, these statements become redundant and should be removed. Traditional backward slicing cannot slice out the statements on which *only* the slicing criteria depend. Namely, other parts of a program might also depend on its sliced out statements. We develop an enhanced program slicing methodology called solo-slicing to slice out the statements on which *only* the slicing criteria depend. Our algorithm is based on worklist and graph search which is shown in Algorithm 1. The key part of the solo slicing algorithm is to recursively update the out-degree of nodes in SDG and delete redundant nodes whose out-degree are zero.

Step 4: Method Definition Delete: After all three procedures above, in this step, we check if it is possible to remove the seed methods definition. If a seed method resides in an application class or a third-party library class, then we remove it. If this method is in Java Runtime Environment (JRE), then we do not remove it.

III. CASE STUDY

We report here briefly our preliminary case studies results. First, we investigate how pervasive cross cutting features are in real world Java programs. We use network connection, database connection, and logging features as the sample of cross cutting features. We analyze 10 programs whose main business logic has nothing to do with the sample features from DaCapo 9.12-bach benchmarks. Among 10 programs, 7 programs have network connection features. 5 programs have database connection features. 5 programs have logging features.

Table I: Call Sites of method *openConnection* and *openStream* in DrJava

SEEDS: java.net.URL.openConnection/openStream
edu.rice.cs.drjava.ui.NewVersionPopup$6.updateAction
edu.rice.cs.drjava.ui.NewVersionPopup.getManualDownloadURL
edu.rice.cs.drjava.ui.MainFrame._generateJavaAPISet
edu.rice.cs.drjava.ui.NewVersionPopup.getBuildTime

Second, we customize DrJava, a lightweight Java programming environment for pedagogic purposes, as a case study. The core functionality of DrJava has little to do with network connection. However, it does have network connections in its code for checking updates. We remove the network feature from DrJava with our proposed method. Network related features are defined by methods *openConnection* and *openStream* in class java.net.URL. Specifically, if the developers want to have network connection in their program, they must call those APIs. Thus, they form the seed methods in our analysis and customization. Table I shows the specific call sites of the seed methods.

Among these call sites, we use the method *updateAction* in class *NewVersionPopup* as an example. We first conduct forward slicing based on the return value of the call site. In this step, 14 sliced out statements are removed. The method openConnection does not have parameters. So the backward solo-slicing will not be performed. At the end, we find out that the method openConnection is in JRE. So the method definition will not be removed. After customization, we test DrJava by the test cases designed by us. First, DrJava can start up successfully. Second, DrJava cannot access Internet to check and download update. Third, rest of functions work normally. The preliminary results show that our approach is a promising method for feature-based software customization.

Acknowledgments: This research is supported in part by the Office of Naval Research (ONR) grant N00014-13-1-0175.

REFERENCES

[1] G. Xu, N. Mitchell, M. Arnold, A. Rountev, and G. Sevitsky, "Software bloat analysis: Finding, removing, and preventing performance problems in modern large-scale object-oriented applications," in *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research (FoSER '10)*, 2010.

[2] Y. Bu, V. Borkar, G. Xu, and M. J. Carey, "A bloat-aware design for big data applications," in *Proc. ISMM '13*, 2013.

[3] G. Xu, "Finding reusable data structures," in *Proc. OOPSLA '12*, 2012.

[4] F. Tip, C. Laffra, P. F. Sweeney, and D. Streeter, "Practical experience with an application extractor for Java," in *Proc. OOPSLA '99*, 1999.