The Reverse Engineering in Oriented Aspect "Detection of semantics clones"

Amel Belmabrouk, Belhadri Messabih

Abstract-Attention to the reverse engineering in oriented aspect programming (AOP) is rapidly growing as its benefits in large software system development and maintenance are increasingly recognized. This paper reports on the challenges of using the reverse engineering in oriented aspect to detect the crosscutting concerns. So we present a new idea to detect a clone semantic in code. We first present the Principe of the AOP, then, we report on application of reverse engineering in legacy industrial software system. The novel aspect of our approach is the use of program dependence graphs (PDGs) wich one of the important techniques of aspect mining to detect duplicate code in programs. We have extended the definition of a code clone to include semantically related code. We reduced the difficult graph similarity problem to a tree similarity problem by mapping interesting semantic fragments to their related syntax.

Keywords: Oriented Aspect programming, Crosscutting Concern, Reverse engineering, Aspect mining, Program Dependence Graphs (PDG's).

I. INTRODUCTION

Software that is used in a real-world environment must change or become less and less useful in that environment. As the context, in wich the software system is deployed, changes, the software has to be maintained and adapted in order to deliver the new functions required of it and to meet the new constraints imposed on it. A necessary pre-requisite for effectively maintaining and evolving a software system is to maintain an "operational" understanding of the system in question, and this is the objective of reverse engineering research.

The landscape of reverse engineering research is now changing in reponse to the evolution of the overall problem. Software architecture extraction is extending to include all the different aspects of software mentioned above.

In this paper, we focus on the challenges of using one of the techniques of aspect mining (reverse engineering) for understanding and debugging the complexity of code that can make maintenance activity easier. We propose the use of program dependence graphs (PDGs).

This article has five sections. The first is a presentation of the use of reverse engineering in aspect oriented programming. We extend, then, aspect mining techniques. We describe, then, our contribution in order to remedy the problem, using a program dependence graph. Finally, we conclude with perspectives and future works.

II. REVERSE ENGINEERING IN ASPECT ORIENTED PROGRAMMING

II.1 Aspect Oriented Programming (AOP)

Aspect Oriented Programming (AOP) is a new programming paradigm, with constructs explicitly devoted to handling crosscutting concerns. In an Object-Oriented system, it often happens that functionalities, such as persistence, exception handling, error management, logging, are scattered across the classes and are highly tangled with the surrounding code portions. Moreover, the available modularization/ encapsulation mechanisms fail to factor them out. Aspects have been conceived to address such situations.

AOP introduces the notion of *aspect*, as the modularization unit for the crosscutting concerns. Common code that affects distant portions of a system can be located in a single module, an aspect.

Aspect Oriented Programming provides explicit constructs for the modularization of the *crosscutting concerns*: functionalities that traverse the principal decomposition of an application and thus cannot be assigned to a single modular unit in traditional programming paradigms. Existing software often contains several instances of such crosscutting concerns such as persistence, logging, caching, etc. Consequently, refactoring of these applications towards AOP is beneficial, as it separates the principal decomposition from these other functionalities, by modularizing the crosscutting concerns.

The process of migrating existing software to AOP is highly knowledge-intensive and any refactoring toolkit should include the user in a change-refine-loop. However, there is considerable room for automation in two respects:

• **Aspect mining** – identification of candidate aspects in the given code and
• **Refactoring** – semantic-preserving transformations that migrate the code to AOP.

II.2 The Reverse Engineering

According to ***Chikofsky***and ***Cross***:

"Reverse engineering is the process of analyzing a subject system with two goals in mind: (1) to identify the system's components and their interrelationships; and, (2) to create representations of the system in another form or at a higher level of abstraction"

III. ASPECT MINING

Aspect mining is typically described as a specialized reverse engineering process, which is to say that legacy systems (source code) are investigated (mined) in order to discover which parts of the system can be represented using aspects. This knowledge can be used for several goals, including reengineering, refactoring, and program understanding.
Since aspect mining is a relatively recent research area, we distinguish different approaches for aspect mining

III.1 Aspect Mining Techniques

A) Fan-In Analysis

The fan-in of a method *M* is defined as the number of calls to method *M* madefrom other methods. Because of polymorphism, one method call can affect thefan-in of several other methods. A call to method *M* contributes to the fan-in of allmethods refined by *M* as well as to all methods that are refining *M*. The moreplaces the method is called from the more likely it is that the method implements acrosscutting concern so the amount of calls (fan-in) is a good measure for theimportance and scattering of the discovered concern. [Mar]

The analysis follows three consecutive steps: (1) Automatic computation of the fan-in metric for all the methods in the targeted source code. The result is stored as a set of "method-callers" structures that can be sorted by fan-in value. (2) Filtering of the results of the first step, by restricting the set of methods to those having a fan-in above a certain threshold; filtering getters and setters from this restricted set. Get/Setters on static fields are not eliminated because these can be used in the Singleton design pattern; filtering utility methods, like toString( ), collections manipulation methods, etc. (3) Analysis of the remaining set of methods. The elements considered at this step are the callers and the call sites, the method's name and implementation, and the comments in the source code. [Mar, 06]

B) Dynamic Analysis

The technique of Formal Concept Analysis (FCA) is fairly simple. Starting from a (potentially large) set of elements and properties of those elements, FCA determines maximal groups of elements and properties, called concepts.

FCA is used for aspect mining according to the following procedure: Execution traces are obtained by running an instrumented version of the program under analysis for a set of use cases. The execution traces associated with the use cases are the objects, while the executed class methods are the attributes. In the resulting concept lattice, the concepts specific of each use case are located, when existing. The use case specific concepts are those labelled by at least one trace for some use case (i.e. the concept contains at least one specific property) while the concepts with zero or more properties as labels are regarded as generic concepts. When the methods that label one concept crosscut the principal decomposition, a candidate aspect is determined.

C) Clustering

Clustering is a division of data into groups of similar objects.
Clustering can be considered the most important unsupervised learning problem: so, as every other problem of this kind, it deals with finding a structure in a collection of unlabeled data.

Many clustering techniques are available in the literature. Most clustering algorithms are based on two popular techniques known as partitional and hierarchical clustering like k-means, fuzzy c-means and hierarchical agglomerative. [Gab, 06]

D) Clone Detection

Clone detection techniques attempt at finding duplicated code, which may have undergone minor changes afterward.
The typical motivation for clone detection is to factor out copy-paste-adapt code, and replace it by a single procedure.

Code clone

A code clone, in general, means a code fragment that has identical or similar code fragments to it in the source code.
However, there is no single or generic definition for a code clone. So far, several methods of code clone detection have been proposed, and each has its own definition of code clone. [Yos, 06]

Clone Types

There are two main kinds of similarity between code fragments.

Fragments can be similar based on the similarity of their program text, or they can be similar based on their functionality (independent of their text). The first kind of clone is often the result of copying a code fragment and pasting into another location. In the following we provide the types of clones based on both the textual (Types 1 to 3) and functional (Type 4) similarities:

Type-1: Identical code fragments except for variations in whitespace, layout and comments.

Type-2: Syntactically identical fragments except for variations in identifiers, literals, types, whitespace, layout and comments.

Type-3: Copied fragments with further modifications such as changed, added or removed statements, in addition to variations in identifiers, literals, types, whitespace, layout and comments.

Type-4: Two or more code fragments that perform the same computation but are implemented by different syntactic variants. [Cha, 07] [Cha, 09] [Mar]

Clone Detection Process

A clone detector must try to find pieces of code of high similarity in a system's source text.

The main problem is that it is not known beforehand which code fragments may be repeated.

Preprocessing

Remover uninteresting code, determine source and comparison units/granularities.

Transformation

One or more extraction and/or transformation techniques are applied to the preprocessed code to obtain an intermediate representation of the code.

Match Detection

Transformed comparison units (and/or metrics calculated for those units) are compared to find similar source units in the transformed code.

Formatting

Clone pair/class locations of the transformed code mapped to the original code base by line numbers and file location.

Post-processing: Filtering

In this post-processing phase, clones are extracted from the source, visualized with tools and manually analyzed to filter out false positives.

Aggregation

In order to reduce the amount of data or for ease of analysis, clone pairs (if not already clone classes) are aggregated to form clone classes or families. [Cha, 09]

Example of code clone

```
1. static void foo() throws RESyntaxException {
2.   String a[] = new String [] { "123,400", "abc", "orange 100" };
3.   org.apache.regexp.REpat = new org.apache.regexp.RE("[0-9,]+");
4.   int sum = 0;
5.   for (inti = 0; i<a.length; ++i)
6.     if (pat.match(a[i]))
7.       sum += Sample.parseNumber(pat.getParen(0));
8.   System.out.println("sum = " + sum);
9. }
10 static void goo(String [] a) throws RESyntaxException {
11.   RE exp = new RE("[0-9,]+");
12.int sum = 0;
13.for (inti = 0; i<a.length; ++i)
14.if (exp.match(a[i]))
15.sum += parseNumber(exp.getParen(0));
16.System.out.println("sum = " + sum);
17. }
```

Sometimes programmers are simply forced to duplicate code because of limitations of the programming language being used. Analyzing these root causes in more detail could help to improve the language design.

Systems are modularized based on principles such as information hiding, minimizing coupling, and maximizing cohesion. In the end—at least for systems written in ordinary programming languages—the system is composed of a fixed set of modules. Ideally, if the system needs to be changed, only a very small number of modules must be adjusted. Yet, there are very different change scenarios and it is not unlikely that the chosen modularization forces a change to be repeated for many modules. The triggers for such changes are called cross-cutting concerns.

Another important root cause is that programmers often reuse the copied text as a template and then customize the template in the pasted context.

Consequences of Cloning

There are plausible arguments that code cloning increases maintenance effort. Changes must be made consistently multiple times if the code is redundant. Often it is not documented where code has been copied. Manual search for copied code is infeasible for large systems. Furthermore during analysis, the same code must be read over and over again, then compared to the other code just to find out that this code has already been analyzed. Only if you make a detailed comparison, which can be difficult if there are subtle differences in the code or its environment, you can be sure that the code is indeed the same. This comparison can be fairly expensive. If the code would have been implemented only once in a function, this effort could have been completely avoided. For these reasons, code cloning is number one on the stink parade of bad smell by Fowler. But there are also counter arguments.

III.2 Clone detection techniques

Clone detection techniques aim at finding duplicated code, which may have been adapted slightly from the original.
Several clone detection techniques have been described and implemented:

- **Textual Approaches**

Textual approaches (or text-based techniques) perform little or no transformation to the 'raw' source code before attempting to detect identical or similar (sequences of) lines of code. Typically, white space and comments are ignored.

- **Lexical Approaches**

Lexical approaches (or token-based techniques) begin by transforming the source code into a sequence of lexical "tokens" using compiler-style lexical analysis. The sequence is then scanned for duplicated subsequences of tokens and the corresponding original code is returned as clones.
Lexical approaches are generally more robust over minor code changes such as formatting, spacing, and renaming than textual techniques.

- **Syntactic Approaches**

Syntactic Approaches (or AST-basedtechniques) use parsers to first obtain a syntactical representation of the source code, typically an abstract syntax tree (AST). The clone detection algorithms then search for similar subtrees in this AST.

- **Semantic Approaches**

Semantics-aware approaches have also been proposed, using static program analysis to provide more precise information than simply syntactic similarity.
In some approaches, the program is represented as a program dependency graph (PDG). The nodes of this graph represent expressions and statements, while the edges represent control and data dependencies. This representation abstracts from the lexical order in which expressions and statements occur to the extent that they are semantically independent. The search for clones is then turned into the problem of finding isomporphicsubgraphs. [Kom]

IV. PROGRAM DEPENDENCE GRAPH

Existing scalable approaches to clone detection are limited to finding program fragments that are similar only in their contiguous syntax. Other, semantics-based approaches are more resilient to differences in syntax, such as reordered statements, related statements interleaved with other unrelated statements, or the use of semantically equivalent control structures. However, none of these techniques have scaled to real world code bases. These approaches capture semantic information from Program Dependence Graphs (PDGs), program representations that encode data and control dependencies between statements and predicates.

A *program dependence graph* (PDG) is a static representation of the flow of data through a procedure. It is commonly used to implement *program slicing*. The nodes of a PDG consist of program points constructed from the source code: declarations, simple statements, expressions, and *control points*. A control point represents a point at which a program branches, loops, or enters or exits a procedure and is labeled by its associated predicate.

Our work describes the algorithm that finds clones semantics. The novel aspect of our approach is the use of program dependence graphs (PDGs)
We introduce an extended definition of code clones, based on PDG similarity that captures more semantic information than previous approaches. We then provide a scalable, approximate algorithm for detecting these clones. We reduce the difficult graph similarity problem to a simpler tree similarity problem by creating a mapping between PDG subgraphs and their related structured syntax.

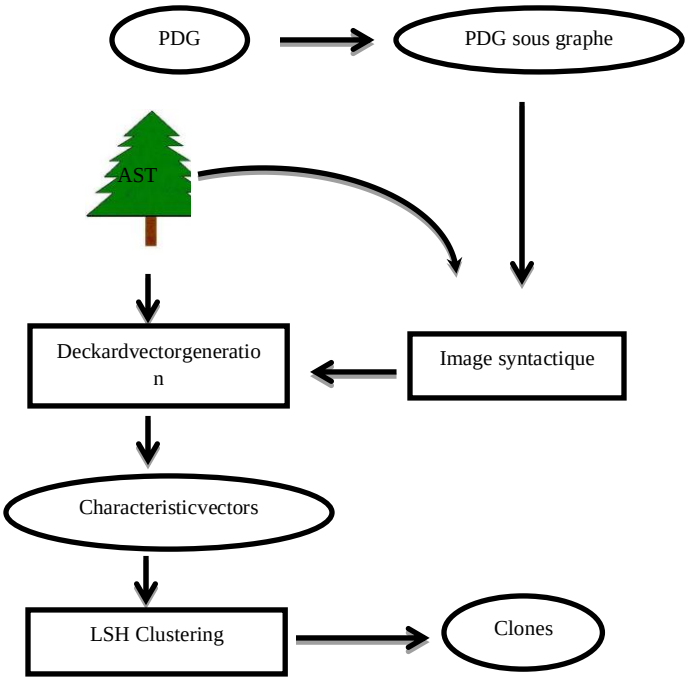

Fig. 1Our **s**emantic clone detection algorithm

High-Level Algorithm

There are difficulties with locating these semantic clones in a scalable manner.

Thus far, no scalable algorithm exists for detecting semantic clones.

We present a scalable, approximate technique for locating semantic clones based on the fact that both structured syntax trees and dependence graphs are derived from the original source code. Because of this relationship, we are able to construct a mapping function that locates the associated syntax for a given PDG subgraph.

We refer to this associated syntax as the *syntactic image*. For compatibility with DECKARD's tree-based clone detection, we map to AST forests.

The syntactic image of a PDG subgraph G, μ(G), is the maximal set of AST subtrees that correspond to the concrete syntax of the nodes in G.

We map each of these nodes to their structured syntax.

Mapping a PDG subgraph to an AST forest effectively reduces the graph similarity problem to an easier tree similarity problem that we can solve efficiently using DECKARD.

Yields something that we can match very efficiently, both partially and fully, using DECKARD's vector generation. This relationship to syntax effectively reduces the graph similarity problem to an easier tree similarity problem.

Our algorithm functions as follows:

1. We run DECKARD's primary vector generation. Subtree and sliding window vectors efficiently provide contiguous syntactic clone candidates for the entire program.
2. For each procedure, we enumerate a finite set of *significant subgraphs*; that is, we enumerate subgraphs that hold semanticrelevance and are likely to be good semantic clone candidates.

In short, we produce subgraphs of maximal size that are likely to represent distinct computations.

3. For each subgraph*G*, we compute $\mu(G)$ to generate an AST forest.
4. We use DECKARD's sliding window vector merging to generate a complete set of characteristic vectors for each AST forest.
5. We introduce *characteristic vectors* to capture structural information of trees (and forests).This is a key step in our algorithm.

Characteristic vectors are generated with a post-order traversal of the parse tree by summing up the vectors for children with the vector for the parent's node.

6. We use LSH to quickly solve the near-neighbor problem and enumerate the clone groups. As before, we apply a set of post-processing filters to remove spurious clone groups and clone group members.

V. CONCLUSION

Different approaches for clone detection have been proposed in the literature. Most of them focus on detecting syntactic similarity of code because checking semantic similarity is very difficult (and in general undecidable).

This paper presents the approach for semantic clone detection based on dependence graphs. We have extended the definition of a code clone to include semantically related code. We reduced the difficult graph similarity problem to a tree similarity problem by mapping interesting semantic fragments to their related syntax.

The main idea of our work is to compute certain *characteristic vectors* to approximate structural information within ASTs and then adapt *Locality Sensitive Hashing* (LSH) to efficiently cluster similar vectors (and thus code clones).

VI. REFERENCES

[1] **[Cha, 07]** Chanchal Kumar Roy and James R. Cordy, A Survey on Software Clone Detection Research, September 26, 2007.

[2] **[Cha, 09]**Chanchal K. Roy, James R. Cordya, Rainer Koschkeb, Comparison and Evaluation of Code Clone Detection Techniques and Tools, Queen's University, Canada University of Bremen, Germany, February 24, 2009.

[3] **[Gab, 06]** GABRIELA .ERBAN AND GRIGORETA SOFIA MOLDOVAN, A COMPARISON OF CLUSTERING TECHNIQUES IN ASPECT MINING, STUDIA UNIV. BABE._BOLYAI, INFORMATICA, Volume LI, Number 1, 2006

[4] **[Kom]**RaghavanKomondoor, Susan Horwitz, Using Slicing to Identify Duplication in Source Code, Computer Sciences Department University of Wisconsin-Madison

[5] **[Mar, 06]** Marius Marin, Leon Moonen and Arie van Deursen, Identifying Crosscutting Concerns Using Fan-in Analysis, Delft University of Technology Software Engineering, 2006

[6] **[Mar]**MagielBruntink, Arie van Deursen, Tom Tourwé, An Evaluation of Clone Detection Techniques for Identifying Crosscutting Concerns

[7] **[Mar]** Marius Adrian MARIN, An Integrated System to Manage Crosscutting Concerns in Source Code

[8] **[Pao]** Paolo Tonella and Mariano Ceccato, Aspect Mining through the Formal Concept Analysis of Execution TracesITC-irst, Centro per la RicercaScienti_ca e Tecnologica

[9] 38050 Povo (Trento), Italy

[10] **[Say]**Syarbaini Ahmad, AbdAzimAbdGhani, Nor FazlidaMohdSani&RodziahAtan, SLICING ASPECT ORIENTED PROGRAM USING DEPENDENCE FLOW GRAPH FOR MAINTENANCE PURPOSE, University Putra Malaysia, Serdang.

[11] **[Yos, 06]** Yoshiki Higo, Toshihiro Kamiya, Shinji Kusumoto, Katsuro Inoue, Method and implementation for investigating code clones in a software system, Graduate School of Information Science and Technology, Osaka University, November 2006.