Aspect Mining Using Method Call Tree

Liping Qu and Daxin Liu
Computer Science and Technology Institute, Harbin Engineering University
Harbin 150001, China
quliping0915@yahoo.com.cn

Abstract

Aspect mining tries to identify crosscutting concerns in legacy systems and thus supports the adaptation to an aspect-oriented design. This paper describes an automatic static aspect mining approach that relies on the method call tree. It uses method call tree to generate method call traces. These traces are then investigated for recurring method patterns based on different constraints, such as the requirement that the patterns exist in always the same composition and in different calling contexts in the method call trace. A experimental evaluation shows that the approach improves the recall of the aspect candidates and the efficiency of the aspect mining.

1. Introduction

A software application is a collection of many different, but related, concerns. In order to ensure its understandability, maintainability and evolvability, developers try to isolate each concern in a separate entity, using structuring techniques offered by the programming language they use. For example, when programming in Java, developers use classes and interfaces as much as possible to represent concerns, and inheritance and aggregation to represent the relations between them.

However, the tyranny of the dominant decomposition [1] implies that no matter how well a software system is decomposed into modular units, some functionality (often called *crosscutting concern*) will always crosscut that decomposition. In other words, such functionality cannot be cleanly captured inside one single module, and consequently its code will be spread throughout other modules.

From a maintenance point of view, such a crosscutting concern is problematic. Whenever this concern needs to be changed, a developer should localize the code that implements it. This may possibly require him to inspect many different modules, since the code may be scattered across several of them. Moreover, identifying the code specifically related to the relevant concern may be difficult. Apart from the fact that the developer may not be familiar with the source code, this code may also be tangled with code implementing other concerns, again due to crosscutting. It should thus come as no surprise that localizing crosscutting code is a time-consuming and error-prone activity.

Aspect-oriented software development (AOSD) has been proposed for solving the problem of the dominant decomposition. Aspect-oriented programming languages add an abstraction mechanism (called an *aspect*) to existing (object-oriented) programming languages. This mechanism allows a developer to capture crosscutting concerns in a localized way.

To apply Aspect-oriented programming to existing systems and transform its crosscutting to proper aspects, a maintainer needs to perform two tasks: one is the identification of crosscutting concerns in the system also known as *aspect mining*, the other is the actual refactoring of such concerns to a localized aspects. Because a maintainer is often not that familiar with all the details of the system, it is desirable to automate these tasks as much as possible.

Although the research area is still in its infancy, several techniques for concern identification have already been developed [2, 3, 4, 5, 6, 7, 8, 9], but most of which are still premature. Silva Breu has proposed an aspect mining using event traces [3], which analyses program traces reflecting the run-time behavior of a system in search of recurring execution patterns. While the approach is generally fairly precise, further analysis reveals several disadvantages: (1) Dynamic program analysis is input-sensitive but incomplete. Dynamic program analysis reasons about really existing behaviors and properties of software systems. The program trace will only hold for a particular set of program inputs. A complete dynamic analysis is not practicable as it is impossible to execute all possible

2007 International Conference on Multimedia and Ubiquitous Engineering(MUE'07)
0-7695-2777-9/07 $20.00 © 2007 IEEE

paths. This means that dynamic analysis even leaves feasible executions unexplored, which would decrease the recall of the aspect candidates. (2) It can happen that some aspect candidates are missed by this approach. Especially in smaller software systems, where different methods are often called only once within a program trace, it can happen that an existing aspect candidate is not identified. For example, method *C* is called by method *A* and method *B*, but method *A* and method *B* exist only once in the sentence if (*condition*) *A*( ); else *B*( );, so method *C* called inside by method *A* or method *B* exists only once in the program traces, and this approach misses method *C* as the aspect candidates. (3) This approach neglects the fact that the different traces have the same part. So, there exists large numbers of repeated calculations in the process of getting candidate aspects, which decreases the efficiency of the aspect mining.

The aim of this paper is to introduce the approach that mines aspects using method call tree. This approach is able to identify crosscutting concerns in software systems with higher recall and efficiency than aspect mining using event traces.

2. Our approach

The basic idea behind aspect mining using method call tree is to observe the source code and create the method call tree in order to get method call traces of the software systems. These traces are then investigated for recurring method call relations. Recurring method call relations are potential crosscutting concerns that describe recurring functionality in the program and thus are possible aspects.

Step 1 create method call tree

Definition 1 Method call tree is a binary tree that describes the method call relations of the software systems. It consists of the control nodes and the method call nodes. The control node marks the type of the sentence where the method is called, including sequence control node, if control code, if-else control node, switch control node, and circle control node. The method call node marks the called method.

Figure 1 shows the structure of the method call tree. The ellipse shows the method call node. The rectangle shows the control node. Subfigure 1.(a) shows the method call node: the name of the method is labeled inside the ellipse. Subfigure 1.(b) shows the sequence control node: left subtree of *Seq* node is the method call node that marks the method called in the sequence sentence, right subtree of *Seq* node is the method call tree that describes the method call relations after this sequence sentence. Subfigure1.(c) shows the if control node: left subtree of *I* node is the method call tree that describes the method call relations while if condition comes into existence, right subtree of *I* node is null, right subtree of *If* node is the method call tree that describes the method call relations after this if sentence. Subfigure1.(d) shows the if-else control node: left subtree of *I* node is the method call tree that describes the method call relations while if condition come into existence, left subtree of *E* node is the method call tree that describes the method call relations while if condition doesn't come into existence, right subtree of *E* node is null, right subtree of *If* node is the method call tree that describes the method call relations after this if-else sentence. Subfigure 1.(e) shows the switch control node: left subtree of *C* node is the method call tree that describes the method call relations while the case expression comes into existence, right subtree of the last *C* node is null, right subtree of *Swi* node is the method call tree that describes the method call relations after this switch sentence. Subfigure 1.(f) shows the circle control node: left subtree of *Cir* node is the method call tree that describes the method call relations while the circle condition comes into existence, right subtree of *Cir* node is the method call tree that describes the method call relations after this circle sentence.

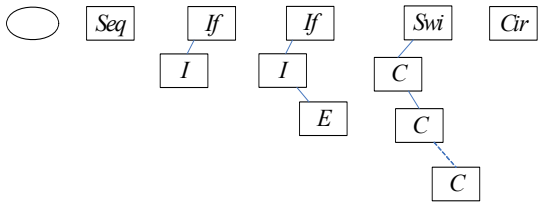

Figure 1. Structure of method call tree

Several method call nodes that mark the same method are permitted to exist in the method call tree. But left subtree of only one node is not null, which describes the method call relations inside the method marked by this method call node.

Step 2 get method call trace

Definition 2 Given a program *P* with a set of method signatures N_P, a method call trace T_P of *P* is a list $t_1t_2,\ldots t_n$ of $t_i \in N_P \times \{ent, ext\} \cup \{\#\}$, where '*ent*' marks entering a method, '*ext*' marks exiting a method, and '×' is Descartes Product of the sets.

The algorithm of getting T_P is defined as follows, where *T* is a method call tree of the software system.

GetTrace(*T*)

1. Initialize stack *S*
2. p = T
3. while p is not null or stack *S* is not empty

4. if p is not null then
5. if p is the method node then
6. output <the method name, *ent*>
7. if p is the control node and $p \neq$ '*Seq*' then
8. output #
9. push p onto stack S
10. p = the root of the left subtree of p
11. else
12. p = pop from stack S
13. if p is the method call node then
14. output <the method name, *ext*>
15. if p is the control node and $p \neq$ '*Seq*' then
16. output #
17. p = the root of right subtree of p

Step 3 get method call relations

Method call relations can be described from the inside and the outside. The outside relation describes the relation between this method and the method that immediately occurs before this method is entered or after this method is left. The inside relation describes the relation between this method and the method that occurs after this method is entered and before this method is left. Now, we introduce four method call relations that are used in this paper.

Definition 3 $\forall A,B \in N_P$, $f=B \rightarrow A$ is called an before relation if <*B*,*ext*><*A*,*ent*> is a sublist of T_P . $S^{\rightarrow}$ is the set of all before relations in T_P.

Definition 4 $\forall A,B \in N_P$, $f=B \leftarrow A$ is called an after relation if <*A*,*ext*><*B*,*ent*> is a sublist of T_P . $S^{\leftarrow}$ is the set of all after relations in T_P.

Definition 5 $\forall A,B \in N_P$, $f=B \bigtriangledown A$ is called an inside-first relation if <*A*,*ent*><*B*,*ent*> is a sublist of T_P. $S^{\bigtriangledown}$ is the set of all inside-first relations in T_P.

Definition 6 $\forall A,B \in N_P$, $f=B \bigtriangleup A$ is called an inside-last relation if <*B*,*ext*><*A*,*ext*> is a sublist of T_P. $S^{\bigtriangleup}$ is the set of all inside-last relations in T_P.

However, to describe *before relation* of the inside first call method and *after relation* of the inside last call method, it is necessary to extend N_P by introduce the designated empty method signature Φ, which represents the absent method. Consequently, the definition of method call relations needs to be extended such that each sublist of a method call trace T_P induces not only the relations defined in Definitions 3 to 6 but also additional relations involving Φ. Table 1 summarizes this extension. The method call relations added by the introduction of Φ are colored in red.

The method call relation matrix $M^{<rel>}$ represents $S^{<rel>}$, <*rel*> $\in \{\rightarrow,\leftarrow,\bigtriangledown,\bigtriangleup\}$. Suppose the number of the method in N_P is m, including Φ, $M^{<rel>}$ is m+1 rank. Rows and columns are marked with method signatures A and B, with A,B$\in N_P \cup \{\Phi\}$. Each matrix entry tells how often a certain relation A<*rel*>B occurs in T_P. Φ represents situations in which there is no method call following or preceding.

Table 1. Extended method call relations

Trace-sublist (N_P)	Relation f ($N_P \cup \{\Phi\}$)
<*B*,*ext*><*A*,*ent*>	$B \rightarrow A$, $A \leftarrow B$
<*A*,*ent*><*B*,*ent*>	$B \bigtriangledown A$, $\Phi \rightarrow B$, $B \leftarrow \Phi$
<*B*,*ext*><*A*,*ext*>	$B \bigtriangleup A$, $B \rightarrow \Phi$, $\Phi \leftarrow B$
#<*A*,*ent*>	$\Phi \rightarrow A$, $A \leftarrow \Phi$
#<*A*,*ext*>	$\Phi \bigtriangleup A$
<*A*,*ent*>#	$\Phi \bigtriangledown A$
<*A*,*ext*>#	$A \rightarrow \Phi$, $\Phi \leftarrow A$
BeginOfList<*A*,*ent*>	$\Phi \rightarrow A$, $A \leftarrow \Phi$, $A \bigtriangledown \Phi$
<*A*,*ext*>*EndOfList*	$A \rightarrow \Phi$, $\Phi \leftarrow A$, $A \bigtriangleup \Phi$

Step 4 get aspect candidates

The method call relations are investigated for recurring method call patterns based on different constraints. We expect that recurring execution patterns are potential crosscutting concerns that describe recurring functionality in the program.

Constraint 1 An method call relation $f=A<rel>B \in S^{<rel>}$ is called uniform if $\forall A \forall B \forall C(A,B,C \in N_P \cup \{\Phi\} \wedge A<rel>B \in S^{<rel>} \wedge C<rel>B \in S^{<rel>}) \Rightarrow A=C$ holds.

Constraint 2 An method call relation $u=A<rel>B \in U^{<rel>}$ is called crosscutting if $\forall A \exists B \exists C(A,B,C \in N_P \wedge A<rel>B \in U^{<rel>} \wedge A<rel>C \in U^{<rel>} \wedge (B \neq C))$ holds, where $U^{<rel>}$ is the set of $f \in S^{<rel>}$ which fulfill *uniform* constraint.

Note that method call relations u that fulfill crosscutting constraint are still called *aspect candidates* as those relations still represent potential crosscutting concerns.

The GetAspect algorithm in the following can get aspect candidates.

GetAspect(M<rel>)
1. for each column of $M^{<rel>}$
2. for each row of $M^{<rel>}$
3. if matrix entry $\neq 0$ then
4. n++;
5. if n=1 and for this entry $A,B \neq \Phi$ then
6. mark $A<rel>B$ with '$*$' in $M^{<rel>}$
7. for each row of $M^{<rel>}$
8. for each column of $M^{<rel>}$
9. if matrix entry is marked with '$*$' then
10. m++;
11. if m>1 then
12. for each entry marked with '$*$'
13. add $A<rel>B$ to aspect candidates set

3. Experimental Evaluation

This section presents an experiment of aspect mining using method call tree. The purpose of our experiment is to verify the efficiency of our approach.

Figure 2 illustrates a outline source code in which the sentences without method call are discarded.

```
main( ){                          D( ){
    B( );                             C( );
    while (condition1){               A( );
        A( );                         B( );
        B( );                         K( );
    }                                 I( );
    F( );                             G( );
    J( );                             E( );
    G( );                         }
    H( );                         E( ){
    A( );                         }
    B( );                         F( ){
    D( );                             K( );
}                                     I( );
A( ){                             }
}                                 G( ){
B( ){                             }
    C( );                         H( ){
    if (condition2)               }
      J( );                       I( ){
    if (condition3) {                 if (condition6)
      G( );                             J( );
      if (condition4)             }
        F( );                     J( ){
      else L( );}                 }
}                                 K( ){
C( ){                             }
    if (condition5) {             L( ){
      G( );                           K( );
      H( );                       }
    }
}
```

Figure 2. A outline source code

Figure 3 shows the method call tree of the source code in figure 2.

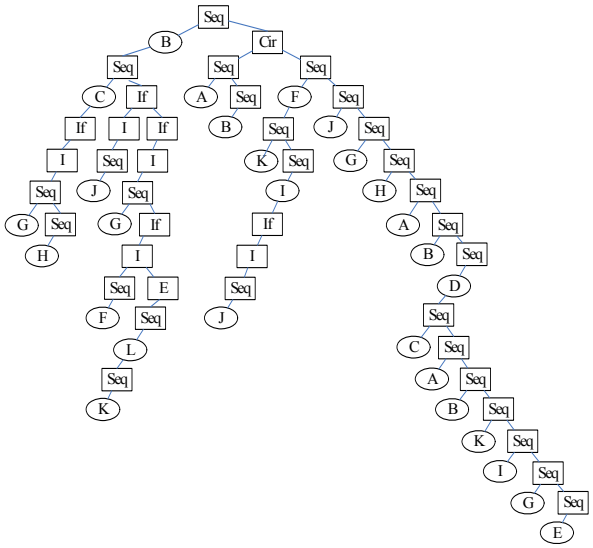

Figure 3. Method call tree of source code

The method call trace T_P can be obtained using GetTrace algorithm: *BeginOfList<B,ent><C,ent>##<G,ent><G,ext><H,ent><H,ext>##<C,ext>##<J,ent><J,ext>####<G,ent><G,ext>##<F,ent><F,ext>##<L,ent><K,ent><K,ext><L,ext>####<B,ext>#<A,ent><A,ext><B,ent><B,ext>#<F,ent><K,ent><K,ext><I,ent>##<J,ent><J,ext>##<I,ext><F,ext><J,ent><J,ext><G,ent><G,ext><H,ent><H,ext><A,ent><A,ext><B,ent><B,ext><D,ent><C,ent><C,ext><A,ent><A,ext><B,ent><B,ext><K,ent><K,ext><I,ent><I,ext><G,ent><G,ext><E,ent><E,ext><D,ext> EndOfList*.

The number of the method existed in figure 2 is 12, so the method call relation matrix $M^{<rel>}$ below as representation of $S^{<rel>}$ are 13 rank. Each matrix entry tells how often a certain relation *A<rel>B* occurs in the method call trace.

$$M^{\rightarrow}=\begin{bmatrix}0&1&1&2&0&0&1&2&0&0&2&2&1\\0&0&3&0&0&0&0&0&0&0&0&0&0\\2&0&0&0&1&0&0&0&0&0&0&1&0\\1&1&0&0&0&0&0&0&0&0&0&0&0\\1&0&0&0&0&0&0&0&0&0&0&0&0\\1&0&0&0&0&0&0&0&0&0&0&0&0\\1&0&0&0&0&0&0&0&0&0&1&0&0\\1&0&0&0&1&0&0&2&0&0&0&0&0\\1&1&0&0&0&0&0&0&0&0&0&0&0\\1&0&0&0&0&0&0&1&0&0&0&0&0\\2&0&0&0&0&0&0&1&0&0&0&0&0\\1&0&0&0&0&0&0&0&0&2&0&0&0\\1&0&0&0&0&0&0&0&0&0&0&0&0\end{bmatrix}\quad M^{\leftarrow}=\begin{bmatrix}0&0&2&1&1&1&1&1&1&2&1&1\\1&0&0&1&0&0&0&0&1&0&0&0&0\\1&3&0&0&0&0&0&0&0&0&0&0&0\\2&0&0&0&0&0&0&0&0&0&0&0&0\\0&0&1&0&0&0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&1&0&0&0&0&0\\1&0&0&0&0&0&0&0&0&0&0&0&0\\2&0&0&0&0&0&0&0&0&1&1&0&0\\0&0&0&0&0&0&0&2&0&0&0&0&0\\0&0&0&0&0&0&0&0&0&0&0&2&0\\2&0&0&0&0&0&1&0&0&0&0&0&0\\2&0&1&0&0&0&0&0&0&0&0&0&0\\1&0&0&0&0&0&0&0&0&0&0&0&0\end{bmatrix}$$

$$M^{\nabla}=\begin{bmatrix}0&0&0&1&0&0&0&0&0&1&0&0&0\\0&0&0&0&0&0&0&0&0&0&0&0&0\\1&0&0&0&0&0&0&0&0&0&0&0&0\\0&0&1&0&1&0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0&0&0&0&0&0\\0&0&0&0&0&0&1&0&0&0&0&0&1\\0&0&0&0&0&0&0&0&0&0&0&0&0\end{bmatrix}\quad M^{\Delta}=\begin{bmatrix}0&0&1&1&0&0&0&0&0&1&0&0&0\\0&0&0&0&0&0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0&0&0&0&0&0\\1&0&0&0&0&0&0&0&0&0&0&0&0\\0&0&0&0&1&0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0&0&0&0&0&0\\0&0&0&0&0&0&1&0&0&0&0&0&0\\0&0&0&0&0&0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0&0&0&0&0&1\\0&0&0&0&0&0&0&0&0&0&0&0&0\end{bmatrix}$$

The aspect candidates set, {G→E, G→H, C▽B, C▽D, K▽F, K▽L}, can be obtained using GetAspect algorithm. Table 2 shows the aspect candidates of source code in figure 2, which are identified by aspect mining using method call tree and event traces.

Table 2 Aspect candidates identified by aspect mining using method call tree and event traces

aspect mining using method call tree	aspect mining using event traces
G→E	G→E
G→H	G→H
C▽B	C▽B
C▽D	C▽D
K▽F	
K▽L	

From table 2 we can see that, aspect mining using method call tree that is proposed in this paper can not only identify all aspect candidates identified by aspect

mining using event traces, such as G→E, G→H, C▽B, and C▽D, but also identify aspect candidates that aspect mining using event traces can not identify, such as K▽F and K▽L. Because K▽F and K▽L exist in execution trace only once, aspect mining using event traces cannot identify them. The method call tree can describe method relations in source code of legacy system generally and truly, so, aspect mining using method call tree improves the recall of the aspect candidates and the efficiency of the aspect mining.

4. Related Work

Several techniques have been proposed for aspect mining. They are based on static program analysis and often require user interaction.

The Aspect Browser[10] identifies crosscutting concerns with textual-pattern matching (much like "grep") and highlights them. The tool assumes that aspects have a signature which can be identified by a textual regular expression. Its success in finding aspect thus strongly depends on naming conventions followed in the analysed program code.

FEAT [11], the Feature Exploration and Analysis Tool is implemented as a plugin for the Eclipse Platform. FEAT visualises concerns in a system using so-called "concern graphs". A concern graph abstracts the implementation details of a concern by storing the structure implementing that concern. This way, it documents explicitly the relationships between the different elements of a concern (classes, fields, methods).

Ophir [12], another framework for automatic aspect mining, identifies initial re-factoring candidates using a control-based comparison. The initial identification phase builds upon code clone detection using program dependence graphs. The next step filters undesirable re-factoring candidates. It looks for similar data dependencies in subgraphs representing code clones. The last phase identifies similar candidates and coalesces them into sets of similar candidates, which are the refactoring candidate classes.

5. Conclusions and Future Work

Our approach of aspect mining using method call tree takes the method call tree into account to recognize aspect candidates. This leads to better mining result.

In future work, we propose to construct the ontology of crosscutting concerns of the congeneric legacy system in order to improve the efficiency of the aspect mining. After crosscutting concerns have been identified, we also propose to build the reusable aspect library and add the crosscutting concerns into aspect library in order to provide the reusable aspect for the new congeneric system.

Aspect mining using method call tree has proved promising in experience. However, in order to draw more general conclusions, it will be necessary to conduct further case studies with programs that have a deep inheritance hierarchy. We also plan to use JhotDraw version 5.4b1, a framework for 2D graphics with ≈ 40 kLOC that is a benchmark for aspect mining techniques, in order to compare the aspect mining using method call tree with other aspect mining techniques.

References

[1] Georg Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin, "Aspect-Oriented Programming", *Proc. of European Conference on Object-Oriented Programming*, 1997.

[2] Elrad T, Filman R E, and Bader A, "Aspect-Oriented Programming", *Communication of the ACM*, 2001, pp. 29-32.

[3] S. Breu and J. Krinke, "Aspect Mining Using Event Traces", *Proc. of the 19th International Conference on Automated Software Engineering*, 2004, pp. 310–315.

[4] P. Tonella and M. Ceccato, "Aspect mining through the formal concept analysis of execution traces", *Proc. of the 11th Working Conference on Reverse Engineering*, 2004, pp. 112–121.

[5] M. Marin, A. van Deursen, and L. Moonen, "Identifying aspects using fan-in analysis", *Porc. of the 11th Working Conference on Reverse Engineering*, 2004, pp. 132–141.

[6] A.Zaidman, T.Calders, "Applying webmining techniques to execution traces to support the program comprehension process", *Proc. of the 8th European Conference on Software Maintenance and Reengineering*, 2004, pp. 329–338.

[7] D. Shepherd, T. Tourw´e, and L. Pollock, "Using language clues to discover crosscutting concerns", *Workshop on the Modeling and Analysis of Concerns*, 2005.

[8] T. Tourw´e and K. Mens, "Mining aspectual views using formal concept analysis", *Proc. of Source Code Analysis and Manipulation Workshop*, 2004

[9] L. He and H. Bai, "Aspect mining using clustering analysis". *Technical report*, Jilin University, 2004.

[10] W.G.Griswold, Y.Kato, and J.J.Yuan, "Aspect Browser: Tool Support for Managing Dispersed Aspects", *Technical Report CS99-0640*, Department of Comuputer Science and Engineering, 1999.

[11] M.P.Robillard and G.c.Murphy, "Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies", *Proc of the 24th international*

Conference on Software Engineering, 2002, pp. 406-416.

[12] D.Shepherd and L.Pollock, "Ophir: A Framework for Automatic Mining and Refactoring of Aspects", *Technical Report 2004-03*, University of Delaware, 2003.