# MINING ANOMALIES IN OBJECT-ORIENTED IMPLEMENTATIONS THROUGH EXECUTION TRACES

Paria Parsamanesh, Amir Abdollahi Foumani
*IBM Rational Software Group*
*Montreal, Quebec, Canada*

Constantinos Constantinides
*Department of Computer Science and Software Engineering, Concordia University*
*Montreal, Quebec, Canada*

Keywords: Software quality, anomalies, program comprehension, aspect mining, dynamic programming.

Abstract: In the context of a computer program, the term "anomaly" is used to refer to any phenomenon that can negatively affect software quality. Examples of anomalies in object-oriented programs include low cohesion of modular units, high coupling between modular units and the phenomenon of crosscutting. In this paper we discuss the theoretical component of a technique to identifying anomalies in object-oriented implementations based on observation of patterns of messages (invoked operations). Our technique is based on the capturing of execution traces (paths) into a relational database in order to extract knowledge of anomalies in the system, focusing on potential crosscutting concerns (aspects). In order to resolve ambiguities between candidate aspects we deploy dynamic programming to identify optimal solutions.

## 1 INTRODUCTION

In the context of object-oriented development, the term "anomaly" refers to any phenomenon that can negatively affect the quality of software. Examples include low cohesion of modular units, and high coupling between modular units. Another example which sometimes lies at the root of the previous two quality problems is the phenomenon of crosscutting, whereby the implementation of certain concerns is not well localized but it cuts across the class hierarchy of the system, resulting in scattering of behavior and tangling of code.

In this paper, we discuss the theoretical component of a technique to identifying anomalies in object-oriented implementations. Our proposal is based on running use-case scenarios and capturing the corresponding execution traces in a relational database where we can then make observations over certain patterns of messages (invoked operations). The knowledge we extract can provide information on potential crosscutting concerns (aspect candidates). In order to identify an optimal solution while choosing an aspect among a collection of aspect candidates, we deploy dynamic programming algorithms.

The rest of this paper is organized as follows: In Section 2 we discuss the necessary theoretical background to this research. In Section 3 we discuss the problem and motivation behind this research. In Section 4 we present our proposal and in Section 5 we present our methodology. In Section 6 we discuss related work and comparisons to our proposal. We conclude our work in Section 7 with a summary, discussion and pointers to future research directions.

## 2 THEORETICAL BACKGROUND

The principle of separation of concerns (Parnas, 1972; Dijkstra, 1976) refers to the realization of system concepts into separate software units and it is a fundamental principle to software development. The associated benefits include better analysis and understanding of systems, readability of code, a high-level of design-level reuse, easy adaptability and good maintainability. To this end, the notions of cohesion and coupling are the fundamental measures of quality in an object-oriented software system. Cohesion is the degree to which a module performs a single responsibility (i.e. it addresses a single concern). Coupling (or dependency) is the degree to which each program module relies on another module. Two objects are said to be coupled if they act upon one another (Chidamber and Kemerer, 1991). While structuring object-oriented programs, aiming for high-cohesion and low coupling are two general goals in order to



Parsamanesh P., Abdollahi Foumani A. and Constantinides C. (2006).
MINING ANOMALIES IN OBJECT-ORIENTED IMPLEMENTATIONS THROUGH EXECUTION TRACES.
In *Proceedings of the First International Conference on Software and Data Technologies*, pages 177-189
DOI: 10.5220/0001321101770189

support separation of concerns, thus providing systems which are easier to understand and maintain.

Despite the success of object-orientation in the effort to achieve separation of concerns, certain properties cannot be directly mapped in a one-to-one fashion from the problem domain to the solution space, and thus cannot be localized in single modular units. Their implementation ends up cutting across the inheritance hierarchy of the system. Crosscutting concerns (or "aspects") include persistence, authentication, synchronization and logging. The "crosscutting phenomenon" creates two implications: 1) the scattering of concerns over a number of modular units and 2) the tangling of code in modular units. As a result, developers are faced with a number of problems including a low level of cohesion of modular units, strong coupling between modular units and difficult comprehensibility, resulting in programs that are more error prone.

Aspect-Oriented Programming (AOP) (Kiczales et al., 1997; Elrad et al., 2001) explicitly addresses those concerns which "can not be cleanly encapsulated in a generalized procedure (i.e. object, method, procedure, API)" (Kiczales et al., 1997) by introducing the notion of an aspect definition, which is a modular unit of decomposition. There is currently a growing number of approaches and technologies to support AOP. One such notable technology is AspectJ (Kiczales et al., 2001), a general-purpose aspect-oriented language, which has influenced the design dimensions of several other general-purpose aspect-oriented languages, and has provided the community with a common vocabulary based on its own linguistic constructs. In the AspectJ model, an aspect definition is a new unit of modularity providing behaviour to be inserted over functional components. This behaviour is defined in method-like blocks called *advice* blocks. However, unlike a method, an advice block is never explicitly called. Instead, it is activated by an associated construct called a *pointcut* expression. A pointcut expression is a predicate over well-defined points in the execution of the program which are referred to as *join points*. When the program execution reaches a join point captured by a pointcut expression, the associated advice block is executed. Even though the specification and level of granularity of the join point model differ from one language to another, common join points in current language specifications include calls to methods and execution of methods. Most aspect-oriented languages provide a level of granularity which specifies exactly when an advice block should be executed, such as executing before, after, or instead of the code defined at the associated join point. Furthermore, much like a class, an aspect definition may contain state and behavior. It is also important to note that AOP is neither limited to object-oriented programming nor to the imperative programming paradigm. However, we will restrict this discussion to the context of object-oriented programming.

## 3 PROBLEM AND MOTIVATION

As the complexity of software systems increases, it becomes more challenging to build software that is free of certain quality problems, collectively referred to as "anomalies." Given an object-oriented program, we would like to be able to achieve program comprehension in order to identify anomalies at specific points over the implementation so that we can then make decisions about possible transformations such as refactoring or reengineering activities.

## 4 PROPOSAL

In this section we discuss our proposal to aid in the provision of high-quality systems. Several techniques have already been proposed in the literature for detecting low cohesion, high coupling and crosscutting in object-oriented implementations (Robillard and Murphy, 2001; Robillard and Murphy, 2002; Krinke and Breu, 2004; Breu and Krinke, 2004; Moldovan and Serban, 2006). In this work we build on current proposals by discussing the theoretical component of a technique to identify anomalies in object-oriented programs based on the monitoring of execution traces and the identification of patterns of invoked operations. More specifically, our proposal is based on running use-case scenarios and capturing the corresponding execution traces into a relational database. We can then apply certain strategies which can help us identify anomalies. In cases of the existence of several candidate aspects, we can deploy dynamic programming to resolve ambiguities.

## 5 METHODOLOGY: EXTRACTING KNOWLEDGE

In this section we discuss how our proposal is being realized.

Consider the relational database schema in Figure 1 which can store message sequences at run time. The main entities of the schema are defined as follows:

- Class: A class encapsulates state and behavior. Class instances (objects) send and receive messages.
- Method: Methods collectively make the behavior of a class. In this model we only consider those methods which appear in execution paths.

- Message: A message is an invocation of a method involving a caller (sender) object and a callee (receiver) object. A message is received and acted upon by a method inside a class definition, i.e. $< message >::= object - name.operation - name$. A message implements a partial responsibility of a system operation and it forms part of a use-case scenario.
- Execution path: A use-case scenario is described as a sequence of interactions between entities. During requirements analysis, this sequence is described at a high-level of abstraction in terms of the interaction between an actor and the system where the latter is viewed as a black box. During design, a fine grained model captures sequences of interactions between class instances through an ordered sequence of message passing. We define this ordered sequence of message passing as an *execution path*, i.e. $< executionpath >::=< message^{+} >$.

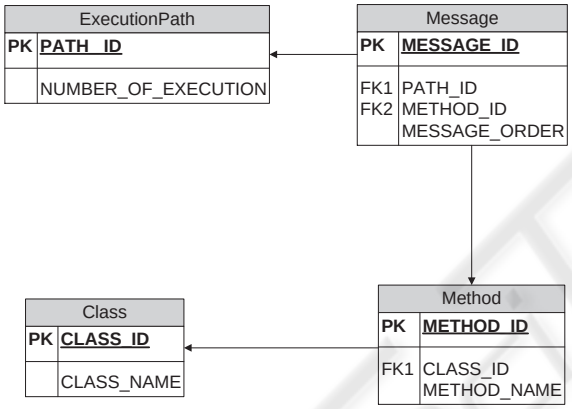


Figure 1: Relational database schema for storing execution paths.

**Main idea**: Our technique to identify anomalies in object-oriented implementations is based on capturing execution paths as data from the runtime environment and populating a relational database. Data can then be analyzed in order to provide information in the form of meaningful statistics. In doing so, we can proceed as follows:

1. Generate data: Execute all possible use-case scenarios. Each use-case scenario is comprised by a sequence of execution paths.
2. Capture data: Capture execution traces as data and store them into a relational database schema.
3. Analyze data: We identify patterns of messages in execution paths. By applying certain strategies which we have developed (to be discussed next), we can identify anomalies based on the result sets.

In the next subsections we will describe the three different types of knowledge on anomalies we wish to acquire from the relational database schema, namely low cohesion, high coupling and crosscuttings. We define algorithms to identify the three types of anomalies.

## 5.1 Identifying Low Cohesion

We provide the following definitions:

**Definition 5.1** *The term "fanin" is used to define the number of messages that a given object receives over a single execution path. The term is also defined for methods.*

**Definition 5.2** *The term "fanout" is used to define the number of messages that a given object sends to other objects over a single execution path. The term is also defined for methods.*

**Definition 5.3** *A class is defined to be independent in the context of a use-case scenario, if 1) the class forms an integral part of the use-case scenario and 2) the class is not dependent on any other classes within the use-case scenario. We can consider independent classes as potential aspects (Foumani and Constantinides, 2005a; Foumani and Constantinides, 2005b).*

We would like to obtain the following knowledge:

- Classes that are involved in (short length) scenarios that have large fanin and low fanout can be considered as having low-cohesion or having an independent role in a use-case scenario.
- Methods that are involved in (short length) scenarios that have large fanin and low fanout can be considered as having low-cohesion or having an independent role in a use-case scenario. These methods can be separated from their original class through refactoring and they can be captured as advice blocks inside aspect definitions.

**Definition 5.4** *We define "class independency factor" ($CIF$) as*

$$CIF = \frac{(class\ fanin)}{(class\ fanout)}$$

In the case where $class\ fanout = 0$ (in which case $CIF \rightarrow \infty$), the class is considered as one with an independent role and it can be a candidate aspect. To calculate this factor we need to execute statements to measure class fanin, and class fanout which is described in the subsequent subsections.

### 5.1.1 Measuring Class Fanin

In the following query we iterate over all messages in all execution paths where a unique identifier for a given class, *CLASS_ID*, is provided, and we count the number of references to the class:

```
Select count(mth.CLASS_ID)
From Message msg
Join Method mth
 On mth.METHOD_ID = msg.METHOD_ID And
 mth.CLASS_ID = [CLASS ID]
```

#### 5.1.2 Measuring Class Fanout

In the following query we count the number of classes referenced by a given class.

```
Select count(Distinct mth_out.CLASS_ID)
From Message msg
Join Message msg_out
  On msg_out.PATH_ID = msg.PATH_ID
  And msg_out.MESSAGE_Id <> msg.MESSAGE_ID
  And msg_out.MESSAGE_ORDER = msg.MESSAGE_ORDER + 1
Join Method mth
  On mth.METHOD_ID = msg.METHOD_ID
  And mth.CLASS_ID = [CLASS ID]
Join Method mth_out
  On mth_out.METHOD_ID = msg_out.METHOD_ID
  And mth_out.CLASS_ID <> [CLASS ID]
```

The queries in 5.1.1 and 5.1.2 enable the calculation of the $CIF$ factor for any given class. We are interested in classes with high $CIF$ values (percentages). These classes can be considered as a bottleneck in an object-oriented system for the following two reasons:

1. They may have many responsibilities (low cohesion). In this case we can follow a refactoring strategy to break the responsibilities and achieve a higher separation of concerns.
2. They may have an independent role (and can be potential aspects).

**Definition 5.5** *We define "method independency factor" (MIF), as*

$$MIF = \frac{(method\ fanin)}{(method\ fanout)}$$

In the case where $method\ fanout = 0$ (in which case $MIF \rightarrow \infty$), the method can be modeled as an advice block inside an aspect definition as this is an independent entity. In order to calculate this factor we need to execute similar queries to those we described above to calculate the fanin and fanout values of each method. We are interested in methods with high $MIF$ values (percentages). These methods can also be considered as a bottleneck for the same reasons described above for classes. As in the case with classes, object-oriented refactoring or migration to an aspect-oriented context can provide a viable solution.

**Definition 5.6** *To identify classes with low cohesion, we define class cohesion factor (CCH),*

$$CCH = \frac{(number\ of\ classes\ in\ execution\ path)}{(number\ of\ methods\ in\ execution\ path)}$$

The $CCH$ factor defines the distribution of invoked methods in an execution path between classes. In other words, $CCH$ defines the distribution of responsibilities between classes. According to this definition we may have the following cases:

- $CCH \ll 1$: Many methods of a few classes are called in a given execution path. This implies that we have classes with low cohesion.
- $CCH \approx 1$: Logical distribution of responsibility between methods of classes. To maintain high cohesion we need to have a $CCH$ value very close to 1. This would imply that the number of invoked methods and the number of classes involved in a use-case scenario should be very close to each other. We can identify an exception when there is only one class and one method referenced in the execution path. In this case, even though $CCH = 1$, we would still have low cohesion of the given class and its corresponding method.

Table 1 provides different illustrative cases for calculating and interpreting $CCH$.

## 5.2 Identifying High Coupling

To investigate coupling and identify classes with high coupling we need to calculate two factors:

1. $CCH$: class cohesion factor. Recall that $CCH \ll 1$ implies that many methods of a few classes are invoked in a given execution path.
2. The fanin of each method. This can be provided by the count of the number of invocations originated from (methods of) other classes.

**Definition 5.7** *For two classes $C_i$ and $C_j$ with methods $M_i$ and $M_j$ respectively, we define "class coupling" factor ($CCP$) as*

$$CCP = \frac{\sum Ci.fanin(M_i, C_j)}{\sum Cj.fanin(M_j, C_i)}$$

*or*

$$CCP = \frac{\sum fanin(Ci, Cj)}{\sum fanin(Cj, Ci)}$$

The term $C_i.fanin(M_i, C_j)$ refers to the number of invocations for method $C_i.M_i$ that are originated from $C_j$. The term $fanin(C_i, C_j)$ refers to the number of messages received in $C_i$ from $C_j$. If $CCP \approx 1$ (the summation of fanin of methods of class $Ci$ is very close to the summation of fanin of methods of class $Cj$) we can say that the degree of coupling between these two classes is high. If one of these summations is zero, then we will have low coupling. If $C_i.fanin(M_i, C_j) \gg 0$, or $Cj.fanin(M_j, C_i) \gg 0$, then we identify another type of anomaly which we call a "high dependency" of $C_j$ over $C_i$ or vice versa. Table 2 provides different illustrative cases for calculating and interpreting $CCP$.

**Example**: Consider Figure 2 which illustrates an example execution path. Initially, we need to obtain $CCH$ to see whether many methods of few classes

Table 1: Examples of class cohesion through the $CCH$ factor with interpretations.

| # of classes | # of methods | CCH | Interpetation |
|---|---|---|---|
| 1 | 1 | 1 | Since this is the only class in the execution path, this implies that the class has low cohesion. |
| 1 | 5 | 0.2 | Low cohesion at the class level. |
| 2 | 20 | 0.1 | Low cohesion at the class level. |
| 4 | 4 | 1 | High cohesion at the class level; Logical distribution of responsibilities between classes. |
| 5 | 7 | 0.7 | High cohesion at the class level. In this case we still may have low cohesion at the method level; To identify low cohesion at the method level we need to calculate the $MIF$ factor for each method. We consider methods with high $MIF$ value as candidate methods with low cohesion. |

are invoked during execution. $CCH$ is obtained as follows:

$$CCH = \frac{2}{8} = 0.25 \ll 1$$

The very small value of $CCH$ indicates a potential high coupling between the involved classes. To investigate high coupling further, we must read the $CCP$ value, which can be obtained as follows:

$$\begin{aligned} \sum C_i.fanin(M_i, C_j) &= fanin(C_i.M_1, C_j) + \\ &\quad fanin(C_i.M_2, C_j) + \\ &\quad fanin(C_i.M_3, C_j) + \\ &\quad fanin(C_i.M_4, C_j) \\ &= 0 + 1 + 1 + 2 \\ &= 4 \end{aligned}$$

$$\begin{aligned} \sum C_j.fanin(M_j, C_i) &= fanin(C_j.M_5) + \\ &\quad fanin(C_j.M_6) + \\ &\quad fanin(C_j.M_7) \\ &= 2 + 1 + 1 \\ &= 4 \end{aligned}$$

$$CCP = \frac{4}{4} = 1$$

The value of $CCP$ indicates that there is indeed high coupling between the two classes $C_i$ and $C_j$.

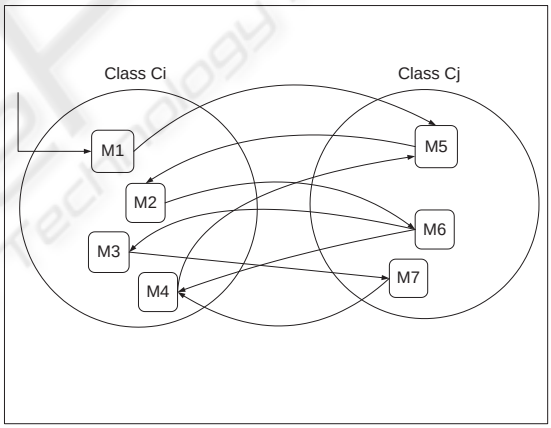


Figure 2: Illustration of high coupling.

## 5.3 Identifying Crosscutting

To identify crosscutting we first need to obtain a set of execution paths. Consider the following:

```
<path1> ::=
  <...,obj2.op2,obj3.op3,obj4.op4,obj5.op5,...>
<path2> ::=
  <...,obj2.op2,obj3.op3,obj4.op4,obj5.op5,...>
<path3> ::=
  <...,obj2.op2,obj3.op3,obj4.op4,obj5.op5,...>
<path4> ::=
  <...,obj1.op1,obj2.op2,...>
<path5> ::=
  <...,obj1.op1,obj2.op2,...>
<path6> ::=
```

Table 2: Interpretation of class dependency.

| $\sum C_i.fanin(M_i, C_j)$ | $\sum C_j.fanin(M_j, C_i)$ | CCP | Interpretation |
|---|---|---|---|
| 0 | $\gg 1$ | $\infty$ | High dependency of $C_i$ over $C_j$. |
| 0 | 1 | $\infty$ | Low coupling. |
| 6 | 7 | $\approx 1$ or $> 1$ | High coupling. |

```
  <...,obj1.op1,obj2.op2,...>
<path7> ::=
  <...,obj4.op4,obj5.op5,obj6.op6,...>
<path8> ::=
  <...,obj4.op4,obj5.op5,obj6.op6,...>
<path9> ::=
  <...,obj3.op3,obj4.op4,obj5.op5,obj6.op6,...>
<path10> ::=
  <...,obj3.op3,obj4.op4,obj5.op5,obj6.op6,...>
```

A crosscutting path, indicated by $S_k$, is one whose sequence of messages occurs in several execution paths. Consider the following crosscutting paths based on the above execution paths $S_{1-10}$:

```
Sa(path 1,2,3):
  <obj2.op2, obj3.op3, obj4.op4, obj5.op5>
Sb(path 4,5,6):
  <obj1.op1, obj2.op2>
Sc(path 7,8):
  <obj4.op4, obj5.op5, obj6.op6>
Sd(path 9,10):
  <obj3.op3, obj4.op4, obj5.op5, obj6.op6>
```

We define $S$ as the set of all sequences $S_k$. Essentially $S$ is a set of candidate aspects. Aspect candidates should not contain common operations because this will violate the semantics (logic) of the initial object-oriented system. As a result, intersections between the aspect candidate sets constitute redundancies. We, therefore, need to identify which one among those sets (or subsets of them) can constitute better candidate aspect sets. In the following subsections we describe an algorithm in order to identify aspect candidates in situations when selecting aspect candidates is ambiguous.

#### 5.3.1 Identifying Orthogonal Aspect Candidate Sets

This subsection presents an algorithm to address the issue of redundancies and the identification of strong candidate aspects. The goal is to eliminate redundancies by identifying aspect candidate sets that do not have common operations. The algorithm has two steps. In the first step the algorithm eliminates the longest common sets from the list of message sequences. The longest common set (there could be more than one) is one which can be obtained by the union of others. As a result, we can remove the longest common sets from $S$, the set of aspect candidates. In the second step the algorithm finds orthogonal sets. An orthogonal set is one which does not have any common element with any other set.

Suppose $S = \{S_1, S_2, S_3, \ldots, S_x, \ldots, S_n\}$ is a set of message sequences and we suppose $S_x$ is a message sequence in the set with the longest common set.

**Step 1**: Eliminating longest common sets. In order to identify a longest common set, we define $S' = S - S_x$. $S_x$ is the longest common set if the following conditions hold:

$$\exists\, S_i \in S' : \;\; S_x \bigcap S_i \neq \emptyset$$

$$\bigcup_{i=1}^{m} S_i = S_x, \;\; \forall\, S_i \in S'$$

The first condition satisfies that a given set has common elements with some of the other message sequences. The second condition satisfies that $S_x$ can be covered by the union of some of the other message sequences. We can now define set $S'$ as a new set of aspect candidates by removing $S_x$ from set $S$. We repeat this process for all the other members of the aspect candidate set until there is no member that can be covered by a union of other members. In the example, we identify $S_d$ as the longest common set (Figure 3), since:

$$S_d \bigcap S_a \neq \emptyset$$

$$S_d \bigcap S_b = \emptyset$$

$$S_d \bigcap S_c \neq \emptyset$$

$$S_d = S_a \bigcup S_c$$

Figure 4 illustrates $S$, the set of all message sequences and set of candidate aspects, after having removed $S_d$, the longest common message sequence.

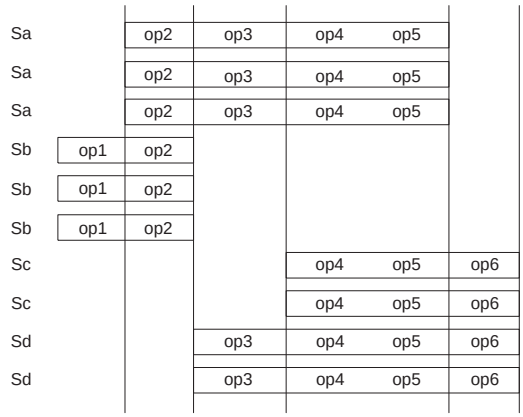


Figure 3: Illustration of execution paths. $S_d$ is the longest common message sequence.

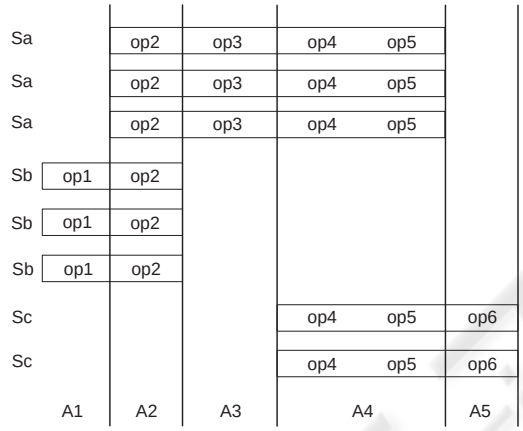


Figure 4: Orthogonal sets.

**Step 2**: Find orthogonal sets. This would imply eliminating the vertical redundancies of operations in Figure 4. We define $S' = S - S_x$ such that $S_x$ is one of the longest message sequence and the following conditions hold:

$$\exists\ S_i \in S' :\ \ S_x \bigcap S_i \neq \emptyset$$

$$\bigcup_{i=1}^{m} S_i \neq S_x,\ \ \forall\ S_i \in S'$$

The first condition satisfies that a given set has common elements with at least one other set. The second condition satisfies that $S_x$ cannot be defined by the union of other sets. We can use the following algorithm to identify orthogonal sets (which are considered aspect candidates):

1. Find common members of $S_x$ with $S_i$ as $C_i$ set

$$\exists\ S_i \in S' :\ \ S_x \bigcap S_i = C_i$$

2. Define $C'_i$ as a set of all members of $S_x$ that are not in $S_i$.

$$S_i - C_i = C'_i,\ \ \forall\ S_i \in S'$$

3. Define $S'_x$ as a set of $S_x$ members except the members that belong to $C_i s$

$$S'_x = S_x - \bigcup_{i=1}^{m} C_i,\ \ \forall\ S_i \in S'$$

4. Replace set $S$ with a new set of aspect candidates such that each $S_i$ in $S$ is replaced by $C_i$ and $C'_i$ and also $S_a$ replace with $S'_a$.

5. We repeat steps 1 to 4 until we have reached the following condition:

$$S_i \bigcap S_j = \emptyset,\ \ \forall\ S_i, S_j \in S\ and\ i \neq j$$

This condition satisfies that there are no two sets with common members in $S$. In this case, the members of $S$ can be defined as aspect candidates.

Applying this algorithm to the example (Figure 4) where $S$ = { $S_a$, $S_b$, $S_c$ }, and starting with sequence $S_a$, we obtain the following:

1. Find common members of $S_a$ with $S_b$ and $S_c$ as $C_b$ and $C_c$ sets:

$$C_b = S_a \bigcap S_b = \{op2\}$$

$$C_c = S_a \bigcap S_c = \{op4, op5\}$$

2. Define $C'_b$ and $C'_c$ as sets of all members of $S_b$ and $S_c$ that are not in $S_a$.

$$C'_b = S_b - C_b = \{op1\}$$

$$C'_c = S_c - C_c = \{op6\}$$

3. Define $S'_a$ as a set of $S_a$ members except the members that belong to $C_b$ and $C_c$

$$S'_a = S_a - (C_b \bigcap C_c) = \{op3\}$$

4. Replace set $S$ with a new set of aspect candidates such that each $S_b$ in $S$ is replaced by $C_b$ and $C'_b$, $S_c$ in $S$ is replaced by $C_c$ and $C'_c$ and $S_a$ replace with $S'_a$.

$$S = \{S'_a, C_b, C'_b, C_c, C'_c\}$$

5. Since $S$ holds orthogonal sets, we do not need to repeat the algorithm.

In Figure 4 we identify $A_{1-5}$ as orthogonal sets, all these being potential aspects. In general, it is highly unlikely that aspects are completely orthogonal of each other. When aspects affect the same join point in one particular sequence, then the order of aspect (advice) execution is important, as this must preserve the semantics of the original object-oriented

system. For example for the aspect candidates in Figure 4, aspects should be defined with the following precedence:

$$A_1 \longrightarrow A_2 \longrightarrow A_3 \longrightarrow A_4 \longrightarrow A_5$$

This algorithm identifies a set of aspects by breaking down the clone message sequences into the lowest level of their method invocations such that there are no conflicts between these sets. However, we believe that this cannot be an optimal solution for the problem since it does not consider the following parameters:

1. Number of identified aspects.
2. Number of different types of concerns addressed by each aspect definition. An aspect definition may be defined with low cohesion.
3. Number of messages in a given execution path.
4. Number of execution paths in a given use-case scenario.
5. Number of runs of all execution paths.

In order to provide optimal solution for the second step of the algorithm, we deploy dynamic programming which is discussed in the next subsections.

#### 5.3.2 Redundancy of Adjacent Invocations (Rai)

In this subsection we introduce a dynamic programming algorithm to identify aspect/advice candidates based on a set of given parameters. Dynamic programming (Cormen et al., 2002) is typically applied to optimization problems. In such problems there can be many possible solutions. Each solution has a value, and we wish to find a solution with the optimal (minimum or maximum) value. In this example, we wish to define parts of message sequences as an aspect (such that the selected operations can be modeled as an advice block) in such a way that the message sequence satisfies the following conditions:

1. It is used in the maximum number of paths.
2. It has the maximum $RAI$ value.
3. It contains the minimum number of classes.
4. It contains the minimum number of operations.

We believe this strategy can identify a stronger set of aspect candidates since it selects the most utilized operations. By minimizing the number of classes and operations we achieve a better modularity. We initially define two new factors to detect clone message sequence and also we introduce new algorithms to identify aspect candidates.

**Definition 5.8** *We define "redundancy of adjacent invocations" ($RAI$) of a given execution path to be the ratio of the recurrence of a sequence $S_k$ of adjacent messages, over the total number of execution paths.*

$$RAI(path) = \frac{(recurrence\ of\ a\ message\ sequence)}{(total\ number\ of\ paths)}$$

As a result, for each execution path in the example of Section 5.3 we have the following $RAI$ measures:

$$RAI(S_a) = \frac{3}{10}$$
$$RAI(S_b) = \frac{3}{10}$$
$$RAI(S_c) = \frac{2}{10}$$

From the set $S = \{S_a, S_b, S_c\}$, we need to select a message sequence or sequences that will be the best candidate for an aspect. Table 3 provides illustrative cases for the following parameters:

NC: Number of classes in a given execution path.

NO: Number of different operations in a given execution path.

NX: Number of execution times of given execution paths.

Table 3: Statistics for message sequences in Figure 3.

| Message Sequence | NC | NO | NX | RAI |
|---|---|---|---|---|
| $S_a$ | 4 | 4 | 1000 | $\frac{3}{10}$ |
| $S_b$ | 2 | 2 | 100 | $\frac{2}{10}$ |
| $S_c$ | 2 | 3 | 100 | $\frac{2}{10}$ |

We define the following recurrence equation to select the next aspect candidate from set $S$ and we define $S'$ as a new set of aspect candidates. Each time we will add $S_x$ that is selected from the following equation into set $S'$.

$$S_x = \begin{cases} S_i & S_i \bigcap S_j = \emptyset,\ \forall\ S_i, S_j \in S. \\ \text{Max(NX),} \\ \text{Max(RAI),} \\ \text{Min(NC),} \\ \text{Min(NO)} & \text{otherwise.} \end{cases}$$

According to this recurrence equation, we select $S_x$ that does not have any common member with the other $S_i$'s in set $S$. For $S_i$'s that have common

members we recursively select $S_x$ that has a maximum $NX$ value, maximum $RAI$ value, minimum $NC$ value and minimum $OP$ value. For the selected $S_x$ we apply the following formula to identify aspect candidates:

$$S_x = S_x - \bigcup_{i=1}^{m} S_i, \ \ \forall\, S_i \in S'$$

In the example, by applying this algorithm we select $S_a$ as the first candidate aspect (as it has $MAX(NX)$). The second aspect candidate according to Table 3 would be $S_b$ and since there is no intersection between $S_a$ and $S_b$, we identify the latter as a new aspect candidate. The last aspect candidate, $S_c$, has an intersection with both $S_a$ and $S_b$. For this reason we apply the above formula for $S_c$ to obtain a strong aspect candidate, i.e.

$$S_c = S_c - (S_a \bigcup S_b) = op3$$

Our RAI optimization strategy has decreased the number of candidate aspects from five down to three.

#### 5.3.3 Family Related Dependency (Frd)

Consider the following paths that contain method invocations in the same class, disregarding any possible gaps between them:

```
<path1> ::=
<...,obj1.op1,...,obj1.op2,...,obj1.op3,...>

<path2> ::=
<...,obj1.op1,...,obj1.op2,...>

<path3> ::=
<...,obj1.op1,...,ob1.op2,...,obj1.op3,...>

<path4> ::=
<...,obj2.op1,...,obj2.op2,...>

<path5> ::=
<...,obj2.op1,...,obj2.op2,...,obj2.op3,...>

<path6> ::=
<...,obj1.op5,...,obj1.op6,...,obj1.op7,...,obj1.op8,...>

<path7> ::=
<...,obj1.op5,...,obj1.op6,...,obj1.op7,...,obj1.op8,...>

<path8> ::=
<...,obj1.op5,...,obj1.op6,...,obj1.op7,...,obj1.op8,...>

<path9> ::=
<...,obj1.op9,...,obj1.op10,...,obj1.op111,...>

<path10> ::=
<...,obj1.op9,...,obj1.op10,...,obj1.op111,...>
```

We define $S_k$ as a sequence of messages that occurs in several paths. For example,

```
Sa(path 1, 3): <obj1.op1, obj1.op2, obj1.op3>
Sb(path 1, 2, 3):<obj1.op1, obj1.op2>
Sc(path 4, 5):<obj2.op1, obj2.op2>
Sd(path 6, 7, 8):<obj1.op5, obj1.op6, obj1.op7, obj1.op8>
Se(path 9, 10):<obj1.op9, obj1.op10, obj1.op111>
```

**Definition 5.9** *We define "family-related dependency" ($FRD$) on a given execution path as the ratio of the recurrence of a sequence $S_k$ of messages sent to the same class, over the total number of execution paths.*

$$FRD(path) = \frac{(number\ of\ selected\ paths)}{(total\ number\ of\ paths)}$$

As a result, for each execution path we have the following $FRD$ measures:

$$FRD(S_a) = \frac{2}{10}$$

$$FRD(S_b) = \frac{3}{10}$$

$$FRD(S_c) = \frac{2}{10}$$

$$FRD(S_d) = \frac{3}{10}$$

$$FRD(S_e) = \frac{2}{10}$$

From the set $S = \{S_a, S_b, S_c, S_d, S_e\}$, we need to select a sequence or sequences that will be the best aspect candidate. Identifying orthogonal aspect algorithm as mentioned before cannot find the optimal set of aspects. In the case of family independent dependency, we introduce another dynamic programming algorithm that uses the following parameters. Table 4 provides illustrative cases for these parameters:

1. Number of different classes in each message sequence ($NC$)(this parameter in $FRD$ is always equal to 1).
2. Number of different operations in each message sequence ($NO$).
3. $FRD$ value for message sequences.

We define the following recurrence equation to select the next aspect candidate from Set $S$ and we define $S'$ as a new set of aspect candidates. Each time we will add $S_x$ that is selected from the following formula into Set $S'$.

$$S_x = \begin{cases} S_i & S_i.obj \notin \bigcup S_j.obj,\ \forall\, S_j \in S, j \neq i. \\ \text{Max(FRD),} \\ \text{Min(NO)} & \text{otherwise} \end{cases}$$

This algorithm selects all the message sequences that use an object that is not used in any other message sequences; For example, $S_c$ in Table 4 uses $obj2$ that is not used in the other message sequences. For the other message sequences we recursively select the

Table 4: Parameters for $FRD$.

| Message Sequence | NC | NO | FRD | obj |
|---|---|---|---|---|
| $S_a$ | 1 | 3 | $\frac{2}{10}$ | obj1 |
| $S_b$ | 1 | 2 | $\frac{3}{10}$ | obj1 |
| $S_c$ | 1 | 2 | $\frac{2}{10}$ | obj2 |
| $S_d$ | 1 | 4 | $\frac{3}{10}$ | obj1 |
| $S_e$ | 1 | 3 | $\frac{2}{10}$ | obj1 |

message sequence with maximum $FRD$ and minimum number of $OP$. For each selected $S_x$ we apply the following formula to identify advice candidates:

$$S_x = S_x - \bigcup_{i=1}^{m} S_i, \ \ \forall\, S_i \in S'$$

In the example, by applying this algorithm we select $S_c$ as the first candidate aspect (as $S_c.obj(= obj2) \notin \bigcup(S.obj - S_c.obj)(= obj1)$). The second aspect candidate according to Table 4 would be $S_b$. We apply the above formula for $S_b$ to obtain a strong aspect candidate:

$$S_b = S_b - S_c = \{obj1.op1, obj1.op2\}$$

As the third aspect candidate we identify $S_d$ that has $MAX(FRD)$:

$$S_d = S_d - (S_c \bigcup S_b)$$

$$= \{obj1.op5, obj1.op6, obj1.op7, obj1.op8\}$$

Nest aspect candidate is $S_a$ or $S_e$ since both has the same $FRD$ and $NO$, In this case, we will select one of the message sequences randomly. for example, we select $S_e$:

$$S_e = S_e - (S_c \bigcup S_b \bigcup S_d)$$

$$= \{obj1.op9, obj1.op10, obj1.op111\}$$

and finally we apply the above formula for $S_a$, since $S_a$ has some common members with $S_b$, we just select the memebers in $S_a$ that are not in $S_b$.

$$S_a = S_a - (S_c \bigcup S_b \bigcup S_d \bigcup S_e)$$

$$= \{obj1.op3\}$$

#### 5.3.4 Combining Strategies: Rai and Frd

In Section 5.3.2 we described that with the calculation of the $RAI$ metric we are able to identify aspects by detecting clones based on method invocations. These methods belong to the different classes and are called one after another in order to play a specific part of the algorithms in the different scenarios. We claim that we can extract these methods, and model them as an aspect. In Section 5.3.3, we discussed how to detect clones based on method invocations such that the methods belong to the same class but can be called in different scenarios in the same order and pattern. Family-related dependency methods can be modeled as different advices of an aspect. In this subsection we combine RAI and FRD. Consider the execution paths illustrated in Table 5 (the result of calculating $RAI$ values) and in Table 6 (the result of calculating $FRD$ values). In Table 7 we show $FRD$ and $RAI$ as well as certain other parameters such as the number of classes, and the number of operations.

```
<path1> ::=
   <..., obj1.op1, obj2.op2,
    ..., obj1.op11,
    ..., obj2.op22, obj1.op111, obj3.op3,
    ...>

<path2> ::=
   <..., obj1.op1, obj2.op2,
    ..., obj2.op22, obj1.op111,
    ...>

<path3> ::=
   <..., obj1.op1, obj2.op2,
    ..., obj1.op11,
    ..., obj2.op22, obj1.op111, obj3.op3,
    ...>
```

We introduce a three-step algorithm to identify aspects in these cases:

**Step 1**: We deploy the $FRD$ algorithm to identify aspects and their corresponding advices. In our example, we identify the following aspect/advices:

```
Aspect{Advice{obj3.op3}}
Aspect{Advice{obj2.op2}}, Advice{obj2.op22}}
Aspect{Advice{obj1.op1}}, Advice{obj1.op111}}
Aspect{Advice{obj1.op11}}
```

**Step 2**: We remove operations selected in the first step from sets identified for $RAI$. In this case we will have three different situations:

1. Selected operations at the beginning of a message sequence in the $RAI$ set. In this case we can remove them from the set. The remaining operations in the set will define a new set.

Table 5: $RAI$ metric for paths.

| Path | Recurring Sequence | RAI |
|---|---|---|
| {path1, path2, path3} | S1:{obj1.op1, obj2.op2} | $\frac{3}{3}$ |
| {path1, path2, path3} | S2:{obj2.op22, obj1.op111} | $\frac{3}{3}$ |
| {path1, path3} | S3:{obj2.op22, obj1.op111, obj3.op3} | $\frac{2}{3}$ |

Table 6: $FRD$ metric for paths.

| Path | Recurring Sequence | FRD |
|---|---|---|
| {path1, path2, path3} | S4:{obj1.op1, obj1.op111} | $\frac{3}{3}$ |
| {path1, path3} | S5:{obj1.op1, obj1.op11, obj1.op11} | $\frac{2}{3}$ |
| {path1, path3} | S6:{obj3.op3} | $\frac{2}{3}$ |
| {path1, path2, path3} | S7:{obj2.op2, obj2.op22} | $\frac{3}{3}$ |

Table 7: $RAI$ & $FRD$ metrics for paths.

| Path | RAI | FRD | NC | NO | Obj |
|---|---|---|---|---|---|
| S1:{obj1.op1, obj2.op2} | $\frac{3}{3}$ | - | 2 | 2 | - |
| S2:{obj2.op22, obj1.op111} | $\frac{3}{3}$ | - | 2 | 2 | - |
| S3:{obj2.op22, obj1.op111, obj3.op3} | $\frac{2}{3}$ | - | 3 | 3 | - |
| S4:{obj1.op1, obj1.op111} | - | $\frac{3}{3}$ | 1 | 2 | obj1 |
| S5:{obj1.op1, obj1.op11, obj1.op11} | - | $\frac{2}{3}$ | 1 | 3 | obj1 |
| S6:{obj3.op3} | - | $\frac{2}{3}$ | 1 | 1 | obj1 |
| S7:{obj2.op2, obj2.op22} | - | $\frac{3}{3}$ | 1 | 2 | obj1 |

2. Selected operations at the end of a message sequence in the $RAI$ set. In this case we can remove them from the set the same as case 1
3. Selected operations are in the middle of a message sequence in the $RAI$ set. In this case we can divide the $RAI$ set into two sets (or more) by removing selected operations from step 1.

**Step 3**: We deploy the $RAI$ algorithm to the new set of message sequences. Here, we stress the importance of maintaining the precedence of aspect execution in order to preserve the semantics of the original object-oriented system.

# 6 RELATED WORK

A number of studies of software metrics such as coupling and cohesion to evaluate the complexity of software have been carried out. However, none of these proposals directly address the calculation software metrics at run-time in order to identify aspects.

In (Gupta and Rao, 2001), the authors measure module cohesion in legacy software. They compared statically calculated metrics against a program execution based approach of measuring the levels of module cohesion. In (Mitchell and Power, 2003), the authors adapt two common object-oriented metrics, coupling and cohesion, and apply them at run-time. In (Moldovan and Serban, 2006), the authors describe a new approach in aspect mining that uses clustering to identify the methods that have the code scattering symptom. In this method, for a method, they consider a large numbers of calling methods and a large numbers of calling classes as indications of code scattering. In order to group the best methods (candidates) they use a vector-space model for defining the similarity between methods. In (Breu and Krinke, 2004), the authors describe an automatic dynamic aspect mining approach which deploys program traces generated in different program executions. These traces are then investigated for recurring execution patterns based on different constraints, such as the requirement that the patterns have to exist in a different calling context in the program trace. In (Krinke and Breu, 2004), the authors describe an automatic static aspect mining approach, where the control flow graphs of a program are investigated for recurring executions based on different constraints, such as the requirement that the patterns have to exist in a different calling context. In (Robillard and Murphy, 2002), the authors introduce a concern graph representation that abstracts the implementation details of a concern and it makes explicit the relationships between different elements of the concern for the purpose of documenting and analyzing concerns. To investigate the practical tradeoffs related to this approach, they developed the Feature Exploration and Analysis tool (FEAT) that allows a developer to manipulate a concern representation extracted from a Java system, and to analyze the relationships of that concern to the code base. In (Robillard and Murphy, 2001), the authors describe concerns based on class members. This description involves three levels of concern elements: use of classes, use of class members, and class member behavior elements (use of fields and classes within method bodies). Use of classes is expressed by the class-use production rules. The rules specify that a concern either uses the entire class to implement its behavior or only part of a class, as well as what parts of the class participate in the implementation of the particular concern. In (Bruntink, 2004), the authors define certain clone class metrics that measure known maintainability problems such as code duplication and code scattering. Subsequently, these clone class metrics are combined into a grading scheme designed to identify interesting clone classes for the purpose of improving maintainability using aspects. In (Baxter et al., 1998), the authors use an abstract syntax tree (AST) to detect duplicated code (clones). This technique uses parsers to first obtain a syntactical representation of the source code, typically an AST. The clone detection algorithms then search for similar subtrees in the AST. In (Jr., 2002), the authors introduce a general-purpose, multidimensional, concern-space modeling schema that can be used to model early-stage concerns.

# 7 CONCLUSION AND FUTURE WORK

In this paper we described the theoretical component of an approach to identifying anomalies in object-oriented implementations based on observations of patterns of messages in a legacy object-oriented system. The term “anomaly” is used to refer to any phenomenon that can negatively affect software quality such as low cohesion, high coupling and crosscutting. Our technique is based on the capture of execution traces (paths) into a relational database in order to extract knowledge of anomalies in the system. We developed strategies to identify anomalies. In the case of ambiguities in the presence of multiple candidate aspects, we deployed dynamic programming to identify optimal solutions in order to group the strongest aspect candidates. We believe that our work can aid developers to find potential anomalies in object-oriented systems. Equally, the work can aid maintainers.

For future work, we intend to provide an implementation through a case study. A tracing mechanism (perhaps deploying AOP) can be used to capture execution traces into a relational database schema.

SQL queries executed over the relational database schema should be used to calculating the parameters of the various algorithms. We also plan to investigate the use of multidimensional schema, data warehouse, and data mining techniques to discover knowledge by considering different dimensions.

## REFERENCES


Baxter, I. D., Yahin, A., Moura, L., Sant'Anna, M., and Bier, L. (1998). Clone detection using abstract syntax trees. In *International Conference on Software Maintenance*.

Breu, S. and Krinke, J. (2004). Aspect mining using event traces. In *IEEE International Conference on Automated Software Engineering*.

Bruntink, M. (2004). Aspect mining using clone class metrics. In *WCRE Workshop on Aspect Reverse Engineering*.

Chidamber, S. R. and Kemerer, C. F. (1991). Towards a metrics suite for object-oriented design. In *Object-Oriented Programming: Systems, Languages and Applications*.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2002). *Introduction to Algorithms*. The MIT Press.

Dijkstra, E. W. (1976). *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, NJ.

Elrad, T., Filman, R. E., and Bader, A. (2001). Aspect-oriented programming. *Communications of the ACM*, 44(10):29–32.

Foumani, A. A. and Constantinides, C. (2005a). Aspect-oriented reverse engineering. In *World Multiconference on Systemics, Cybernetics and Informatics*.

Foumani, A. A. and Constantinides, C. (2005b). Reengineering object-oriented designs by analyzing dependency graphs and production rules. In *IASTED International Conference on Software Engineering and Applications*.

Gupta, N. and Rao, P. (2001). Program execution based module cohesion measurement. In *International Conference on Automated Software Engineering*.

Jr., S. M. S. (2002). Early-stage concern modeling. In *AOSD Workshop on Early Aspects*.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. (2001). An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355.

Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. (1997). Aspect-oriented programming. In Akşit, M. and Matsuoka, S., editors, *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP)*, volume 1241, pages 220–242. Springer-Verlag.

Krinke, J. and Breu, S. (2004). Control-flow-graph-based aspect mining. In *WCRE Workshop on Aspect Reverse Engineering*.

Mitchell, A. and Power, J. F. (2003). Toward a definition of run-time object-oriented metrics. In *ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering*.

Moldovan, G. S. and Serban, G. (2006). Aspect mining using a vector-space model based clustering approach. In *AOSD Workshop on Linking Aspect Technology and Evolution Revisited*.

Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15:1053–1058.

Robillard, M. P. and Murphy, G. C. (2001). Analyzing concerns using class member dependencies. In *ICSE Workshop on Advanced Separation of Concerns in Software Engineering*.

Robillard, M. P. and Murphy, G. C. (2002). Concern graphs: Finding and describing concerns using structural program dependencies. In *International Conference on Software Engineering*.