Towards the Identification of Cross-Cutting Concerns: A Comprehensive Dynamic Approach Based on Execution Relations

Dongjin YU†(a), Xiang SU†, and Yunlei MU†, Members

SUMMARY Aspect-oriented software development (AOSD) helps to solve the problem of low scalability and high maintenance costs of legacy systems caused by code scattering and tangling by extracting cross-cutting concerns and inserting them into aspects. Identifying the cross-cutting concerns of legacy systems is the key to reconstructing such systems using the approach of AOSD. However, current dynamic approaches to the identification of cross-cutting concerns simply check the methods’ execution sequence, but do not consider their calling context, which may cause low precision. In this paper, we propose an improved comprehensive approach to the identification of candidate cross-cutting concerns of legacy systems based on the combination of the analysis of recurring execution relations and fan-ins. We first analyse the execution trace with a given test case and identify four types of execution relations for neighbouring methods: exit-entry, entry-exit, entry-entry and exit-exit. Afterwards, we measure the methods’ left cross-cutting degrees and right cross-cutting degrees. The former ensures that the candidate recurs in a similar running context, whereas the latter indicates how many times the candidate cross-cuts different methods. The final candidates are then obtained from those high fan-in methods, which not only cross-cut others more times than a predefined threshold, but are always entered or left under the same running context. The experiment conducted on three open source systems shows that our approach improves the precision of identifying cross-cutting concerns compared with tradition ones.

key words: cross-cutting concerns, aspects, execution relations, execution trace, running context, fan-in

1. Introduction

During the process of software development, requirements are mapped to software entities which in turn implement them. Object-oriented programming makes such mapping straightforward because those real-world entities in the application domain are mapped to a hierarchy of classes, around which the software is developed. However, not every requirement can be easily mapped to a single modular unit (class) due to the limitation of object-oriented programming itself. When the implementation of a requirement is distributed across multiple classes (scattering) and mixed with the main functionalities (tangling), we call them cross-cutting concerns mixed with business concerns. It is usually hard to maintain such a software system due to its code’s scattering and tangling.

Aspect-oriented software development (AOSD) packages the cross-cutting concerns, which show up in the whole system and have less association with the core business, into a reusable aspect [1]. By separating cross-cutting concerns and business concerns, the aspects can be automatically waved to the object-oriented software system. In this way, it makes software easier to understand and maintain, as well as improving its modularity and stability [2].

Identifying the cross-cutting concerns of legacy systems is the key to reconstructing such systems using the approach of AOSD [3]. Cross-cutting concerns usually share structural or behavioural characteristics. For example, the logging methods are always invoked immediately after the methods that implement the business logic begin execution, whereas a transaction is always activated and committed or rolled back before and after fetching records from a database. Many researchers currently focus on approaches to the identification of cross-cutting concerns in legacy systems to help with code refactoring. There are two kinds of approaches based on whether the programs are running or not [4]. Static code analysis, like syntax and type analysis, targets the source code and the object code (the byte code) and even the design model [5]–[7]. Dynamic analysis, on the other hand, obtains internal information and the output by running the specific program to find relations [3], [8], [9]. Formal concept analysis, as an example of static analysis approaches, starts from a (potentially large) set of elements and properties of those elements. It determines maximal groups of elements and properties, which are called ‘concepts’. Every such concept consists of a set of elements that have one or more properties in common, such that no other elements have those properties nor are there any other declared properties that they have in common. The approach based on event traces, as an example of dynamic analysis approaches, use program traces that are generated in different program executions as an underlying data pool. These traces are then investigated for recurring execution patterns based on different constraints, such as patterns having to exist in different calling contexts in the program trace. Some other approaches have also been proposed [10]–[13].

However, the current approaches mentioned above do not seem particularly efficient. The concept-based approach analyses identifies depending on good naming conventions. However, different programmers may name the classes, methods and variables according to their own preferences. Moreover, the concept-based approach will be confronted with its exponential growth as the number of concept lattice grows bigger. On the other hand, the approach using event traces identifies recurring execution patterns through all the methods that are called, regardless of the possible situations...
according to which some of the methods cannot represent cross-cutting concerns. This may bring about a great deal of noise and, therefore, reduce precision.

Distinct from the existing works mentioned above, we identify the candidate cross-cutting concerns based on a comprehensive approach that discovers recurring execution relations and measures the methods’ fan-ins dynamically. It first collects the execution traces with a given test case and counts the methods’ fan-ins by running the test case. Afterwards, the execution traces of the candidate methods are further investigated to discover the execution relations, which are categorized into four types - exit-entry, entry-exit, entry-entry and exit-exit - by considering the sequence of methods’ invocation. Next, it checks the methods’ left cross-cutting degrees and right cross-cutting degrees. The former ensures that the candidate recurs in a similar context, whereas the latter indicates how many times the candidate cross-cuts different methods. The final candidates are obtained from those high fan-in methods, which not only cross-cut others more times than a predefined threshold, but are always entered or quitted under the similar context. The proposed approach has no constraint about coding conventions as compared with that based on static identifies. Moreover, it improves the precision compared with the traditional approaches based on trace analysis, such as that presented in paper [8].

The remainder of this paper is structured as follows. After introducing the definitions of execution relations in Sect. 2, we present the fan-in metric and the cross-cutting degrees used to filter out some false candidates in Sects. 3 and 4 respectively. In Sect. 5, we describe the process of identifying candidate cross-cutting concerns in detail. The experimental results are given in Sect. 6, followed by the related works in Sect. 7. In last section, we draw conclusions and outline future work.

2. Execution Relations

A program trace is a sequence of methods for invocations and exits. We focus on method executions because those methods with certain recurring execution patterns may indicate the features of cross-cutting concerns.

Definition 1: We call the entry or exit of a method a method point when running a program, denoted by $t_i = (u, entry)$ or $t_j = (u, exit)$ where $(u, entry)$ means entering method $u$ and $(u, exit)$ means exiting method $u$.

Definition 2: A sequence of execution traces consists of all method points involved in running a program, denoted by $S_i = [t_1, t_2, \ldots, t_{i-1}, t_i, \ldots, t_n]$, in which $t_i$ executes after $t_{i-1}$, or $t_{i-1} \Rightarrow t_i$.

The sub-sequences of running execution traces may exhibit similar patterns. We find that if method $u$ should be a cross-cutting one, it may occur in four cases, or executes relations: 1) It executes immediately before other methods begin to execute (Fig. 1 (a)). 2) It executes immediately after other methods finish execution (Fig. 1 (b)). 3) It executes immediately after the other methods begin to execute (Fig. 1 (c)). 4) It executes immediately before other methods finish execution (Fig. 1 (d)). Consequently, we define four types of execution relations according to the different sequences of method points as follows.

Definition 3: For the execution trace $S_i$, if the execution enters the method $v$ immediately after exiting the other method $u$, the method $u$ is said to have a neighbouring exit-entry relation with the method $v$, represented as $(u, exit) \Rightarrow (v, entry)$, or simply $u \rightarrow v$. All the neighbouring exit-entry relations for the execution trace $S_i$ form a neighbouring exit-entry relation set, denoted by $R_{exit}^i$, where $R_{exit}^i = \{u \rightarrow v | (u, exit) \Rightarrow (v, entry), (u, exit) \in S_i, (v, entry) \in S_i\}$.

Definition 4: For the execution trace $S_i$, if the execution enters the method $u$ immediately after exiting the other method $v$, the method $u$ is said to have a neighbouring entry-exit relation with the method $v$, represented as $(v, entry) \Rightarrow (u, exit)$, or simply $u \leftarrow v$. All the neighbouring entry-exit relations for the execution trace $S_i$ form a neighbouring entry-exit relation set, denoted by $R_{entry}^i$, where $R_{entry}^i = \{u \leftarrow v | (u, exit) \Rightarrow (v, entry), (u, exit) \in S_i, (u, entry) \in S_i\}$.

The method $u$ has a neighbouring entry-exit relation with the method $v$, which also means that the method $v$ has a neighbouring exit-entry relation with the method $u$. In other words, $R_{exit}^i$ and $R_{entry}^i$ can be deduced from each other.

Definition 5: For the execution trace $S_i$, if the execution enters the method $u$ immediately after entering the other method $v$, the method $u$ is said to have a neighbouring entry-entry relation with the method $v$, represented as $(v, entry) \Rightarrow (u, entry)$, or simply $u \land v$. All the neighbouring entry-entry relations for the execution trace $S_i$ form a neighbouring entry-entry relation set, denoted by $R_{entry}^i$, where $R_{entry}^i = \{u \land v | (u, entry) \Rightarrow (v, entry), (v, entry) \in S_i, (u, entry) \in S_i\}$.

Definition 6: For the execution trace $S_i$, if the execution exits the method $v$ immediately after exiting the other method $u$, the method $u$ is said to have a neighbouring exit-exit relation with the method $v$, represented as $(u, exit) \Rightarrow (v, exit)$, or simply $u \lor v$. All the neighbouring exit-exit relations for the execution trace $S_i$ form a neighbouring exit-exit relation set, denoted by $R_{exit}^i$, where $R_{exit}^i = \{u \lor v | (u, exit) \Rightarrow (v, exit), (u, exit) \in S_i, (v, exit) \in S_i\}$.

Figure 2 shows a code fragment and its corresponding execution trace and execution relation sets if the execution begins sequentially from Line 1 to Line 18.
To the scale of the legacy system, we set a proper fan-in constraint. Sometimes, it is possible that it be the candidate cross-cutting concern.

Definition 7: The fan-in of the method \( u \) in the execution trace \( S_t \) is denoted by \( F(u, S_t) \), where \( F(u, S_t) = |\{(u, \text{entry})|(u, \text{entry}) \in S_t\}| \). Here, \( |\ast| \) means the number of elements in the collection.

Some methods that are invoked to fulfill operations like persistence, transactions, permission checks and exception handling may come across multiple business modules. These methods might be identified as cross-cutting concerns if they are frequently revoked in different modules. According to the scale of the legacy system, we set a proper fan-in threshold \( T_{fan-in} \). For the one that has a greater or equal fan-in than the threshold \( T_{fan-in} \), or is revoked at least \( T_{fan-in} \) times, it is possible that it be the candidate cross-cutting concern.

### 4. Measuring the Cross-Cutting Degrees

The recurring execution patterns that appear in different contexts might be the cross-cutting concerns. In other words, each cross-cutting concern has two following features: 1) it cross-cuts several distinct methods; 2) similar contexts always lead to its occurrence in one way or another. The following presents the definitions for the so-called cross-cutting degrees used to measure the probabilities for the method to be a cross-cutting concern.

#### 4.1 Left Cross-Cutting Degrees

The calls of such methods as logging, permission checking and exception handling generally occur in similar ways. For example, the logging methods always execute before the others begin to execute. The persistent methods are usually close to that of a database connecting and closing. According to the different recurring patterns, we define four recurring constraints, as follows.

**Definition 8.1:** Given the method \( v \) and the execution trace \( S_t \), the left cross-cutting degree of \( v \) for exit-entry relations is denoted by \( LCD_L^e(v) \), where \( LCD_L^e(v) = |\{(u^e \rightarrow v)|(u^e \rightarrow v) \in R_L^e\}| \). Here, \( |\ast| \) means the distinct number of elements in the set.

Similarly, we define the left cross-cutting degrees for the other three execution relations.

**Definition 8.2:** Given the method \( v \) and the execution trace \( S_t \), the left cross-cutting degree of \( v \) for exit-exit relations is denoted by \( LCD_L^x(v) \), where \( LCD_L^x(v) = |\{(u^e \rightarrow v)\in R_L^x\}| \).

**Definition 8.3:** Given the method \( v \) and the execution trace \( S_t \), the left cross-cutting degree of \( v \) for entry-exit relations is denoted by \( LCD_L^e(v) \), where \( LCD_L^e(v) = |\{(u^e \leftarrow v)\in R_L^e\}| \).

**Definition 8.4:** Given the method \( v \) and the execution trace \( S_t \), the left cross-cutting degree of \( v \) for entry-exit relations is denoted by \( LCD_L^e(v) \), where \( LCD_L^e(v) = |\{(u^e \leftarrow v)\in R_L^e\}| \).

As for the example given in Fig. 2, \( LCD_L^e(a) = 0 \), \( LCD_L^e(c) = 2 \), \( LCD_L^e(d) = 1 \) and \( LCD_L^e(h) = 1 \).

For the method \( a \) to be a cross-cutting concern, the left cross-cutting degrees of the method \( a \) that method \( u \) crosscuts should be equal to 1, which just means that the method \( v \) always leads the same occurrences, i.e., the entry or exit, of method \( u \).

#### 4.2 Right Cross-Cutting Degrees

In addition, to be repeatedly invoked, the candidate concerns should appear in different locations but with similar contexts. The following gives four kinds of cross-cutting constraints that the candidate concern should satisfy.
Definition 9.1: Given the method \( u \) and the execution trace \( S_t \), the **right cross-cutting degree** of \( u \) for exit-entry relations is denoted by \( RCD_{\text{r}}^S(u) \), where \( RCD_{\text{r}}^S(u) = \|((u \rightarrow v')(u \leftarrow v')) \in R_{\text{r}}^S\| \).

Similarly, we define the **right cross-cutting degrees** for the other three execution relations.

Definition 9.2: Given the method \( u \) and the execution trace \( S_t \), the **right cross-cutting degree** of \( u \) for entry-exit relations is denoted by \( RCD_{\text{r}}^S(u) \), where \( RCD_{\text{r}}^S(u) = \|((u \leftarrow v')(u \rightarrow v')) \in R_{\text{r}}^S\| \).

Definition 9.3: Given the method \( u \) and the execution trace \( S_t \), the **right cross-cutting degree** of \( u \) for entry-entry relations is denoted by \( RCD_{\text{r}}^S(u) \), where \( RCD_{\text{r}}^S(u) = \|((u \leftarrow v')(u \leftarrow v')) \in R_{\text{r}}^S\| \).

Definition 9.4: Given the method \( u \) and the execution trace \( S_t \), the **right cross-cutting degree** of \( u \) for exit-exit relations is denoted by \( RCD_{\text{r}}^S(u) \), where \( RCD_{\text{r}}^S(u) = \|((u \rightarrow v')(u \rightarrow v')) \in R_{\text{r}}^S\| \).

As for the example given in Fig. 2, \( RCD_{\text{r}}^S(e) = 2 \), \( RCD_{\text{r}}^S(g) = 1 \), \( RCD_{\text{r}}^S(e) = 3 \) and \( RCD_{\text{r}}^S(j) = 2 \).

The bigger the **right cross-cutting degree** that the method \( u \) has, the more likely it is to be a cross-cutting concern. We define the cross-cutting threshold, \( T_{\text{cross}} \). Only those methods whose **right cross-cutting degrees** are bigger than or equal to \( T_{\text{cross}} \) are considered to be candidate cross-cutting concerns.

5. Identification of Candidate Cross-Cutting Concerns

Since the cross-cutting concerns always occur in a similar context, now and then, we try to identify the candidate ones from the execution traces for a given test case that covers the main routine of the program. Figure 3 shows the overall process of the approach to the identification of cross-cutting concerns, which is further divided into the following three steps.

**Step 1: Catch the execution traces for a given test case**

In this step, the sequence of execution traces for a given test case is captured and, meanwhile, the fan-ins of the invoked methods are counted. The only things we need to do are as follows: 1) Write an AspectJ file, as Table 1 illustrates, for the software whose crosscutting concerns need to be mined; 2) Convert the software to an AspectJ project; 3) Choose a test case and run it. Here, the test case refers to a set of conditions or variables under which a tester can fulfill one certain routine using the software in order to capture the running sequences of entries and exits of each method, or the execution traces.

Table 1 shows the piece of code of the corresponding AspectJ file as an example for Step 1. Here, the set of pairs of the method’s signature and its fan-in is denoted by \( R_{\text{fan-in}}^{S_j} \). In other words, \( R_{\text{fan-in}}^{S_j} = \{(u, F(u, S_j))\} \).

**Step 2: Identify the execution relations**

**Step 3: Choosing Candidate Cross-cutting Concerns**

After obtaining the execution relations in Step 2, we choose the methods that are entered or quit in the different places but with similar contexts as the candidate cross-cutting concerns. Moreover, we only consider those methods with high fan-ins. Table 3 shows the algorithm used to choose candidate cross-cutting concerns from execution
relations and methods' fan-ins.

First, in $R^S_v, R^S_u, R^K_v$ and $R^K_u$, we filter out the execution relations such as $u \rightarrow v$, $u \leftarrow v$, $u \land v$ and $u \lor v$, in which $u$'s right cross-cutting degree is below the predefined threshold $T_{cross-cutting}$, or where $u$ has a smaller fan-in than $T_{fan-in}$ (Lines 1 to 4). Meanwhile, in $R^S_v, R^S_u, R^K_v$ and $R^K_u$, we choose the execution relations such as $u \rightarrow v$, $u \leftarrow v$, $u \land v$ and $u \lor v$, in which $u$'s left cross-cutting degree is equal to 1 (Lines 5 to 8). The final set of candidate cross-cutting concerns $R_{aspect}$ can then be obtained from the above-chosen execution relations (Line 9). They are the high fan-in methods, which not only cross-cut others at least $T_{cross-cutting}$ times, but which are always entered or quitted in the similar context.

### 6. Experiments

In order to evaluate the effectiveness of our approach, we developed a tool called the Candidate Cross-cutting Concern Identification Tool (C3IT) to investigate the execution traces that are caught using AJDT 2.2.1, an eclipse platform-based tool support for AOSD with AspectJ. We ran our experiments on Windows 7 with Intel core2 Duo CPU E7500. Figure 5 shows a screenshot of C3IT.

We conducted three experiments with PetStore, JHotDraw and Telecom using C3IT. We compared our approach with that of paper [8] in terms of the metrics of recall and precision. Here, the precision is the fraction of retrieved instances that are relevant, while the recall (also known as the ‘sensitivity’) is the fraction of relevant instances that are retrieved. A more specific introduction to evaluating the algorithms for aspect identification can be found in paper [15] and [16].

**PetStore.** Java PetStore is a sample J2EE e-business application developed by Oracle. It is intended as a demonstration of a real life web application that allows customers to purchase via a web browser. Petstore includes 40 java files in seven packages and has approximately 3,800 non-comment lines of code. In order to reduce the number of unconcerned methods, we set $T_{fan-in}$ as 5 and $T_{cross-cutting}$ as 2. We use ControllerServlet.doGet and ControllerServlet doPost as the entry points of the execution trace for the test case because nearly all requests will be handled after these two methods are invoked. The test scenario is as follows: first “catalogue” the pets then “search”
“Sweet Parrot” and finally “buy” it.

**JHotDraw.** JHotDraw is an application framework for two-dimensional graphics. It was designed as an exercise to show a good use of design patterns. We use JHotDraw 6.0 b1, which includes 484 java files in 10 packages and has approximately 28,360 non-comment lines of code. JavaDrawApp.main is executed as an entry point because most of the components will be covered if running from it. The test traverses such packages as figures, contrib, framework, standard, util and their sub-packages, excluding the example and test sub-packages. The test scenario is as follows. We first “create” an edit box and then “draw” a rectangle, a circle and a triangle in the edit box. Afterwards, we “undo” twice and “redo” twice. Finally, we “save” the file. In order to reduce the number of unconcerned methods and let it be comparable with the case of PetStore, we set $T_{fan-in}$ as 5 and $T_{cross-cutting}$ as 2.

**Telecom as an Example of AspectJ.** In this case, we want to check whether this approach can detect cross-cutting concerns in java programs that are already extended by aspects written in AspectJ. For that purpose, Telecom example from AspectJ distribution has been chosen, which is a package including 10 java files that have approximately 220 non-comment lines of code. It simulates customers making telephone calls with three different connection types, among which BasicSimulation just performs the calls with basic functionality whereas TimingSimulation is the extension of BasicSimulation with a timing aspect that keeps track of a connection’s duration and cumulates a customer’s connection durations. BillingSimulation is another extension to BasicSimulation with a billing aspect that adds functionality to calculate charges for the phone calls of each customer based on connection type and duration. In order to cover the methods that might be cross-cutting concerns, we choose the main method of BillingSimulation as the entry to the test. Considering the scale of implementation and complexity of the running program, we set $T_{fan-in}$ as 5 and $T_{cross-cutting}$ as 2.

Due to limited space, we only present some intermediate results in Table 4 for Telecom, which is a relatively small-sized example. Table 5 shows the cross-cutting concerns identified from all three cases.

---

**Table 4** Number of different execution relations for Telecom.

<table>
<thead>
<tr>
<th>Execution relation set</th>
<th>$R_{E}^{1}$</th>
<th>$R_{E}^{2}$</th>
<th>$R_{E}^{3}$</th>
<th>$R_{E}^{4}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of execution relations</td>
<td>26</td>
<td>27</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Execution relation set</td>
<td>$R_{E}^{1}$</td>
<td>$R_{E}^{2}$</td>
<td>$R_{E}^{3}$</td>
<td>$R_{E}^{4}$</td>
</tr>
<tr>
<td>Number of execution relations</td>
<td>11</td>
<td>7</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Execution relation set</td>
<td>$R_{E}^{1}$</td>
<td>$R_{E}^{2}$</td>
<td>$R_{E}^{3}$</td>
<td>$R_{E}^{4}$</td>
</tr>
<tr>
<td>Number of execution relations</td>
<td>11</td>
<td>9</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

**Table 5** The identified candidate cross-cutting concerns from three cases.

<table>
<thead>
<tr>
<th>Project</th>
<th>Identified methods as cross-cutting concerns</th>
</tr>
</thead>
<tbody>
<tr>
<td>PetStore</td>
<td>EntityManager.createQuery(), Query.setParameter(), Item.getTotalScore(), PetstoreUtil.closeIgnoringException()</td>
</tr>
<tr>
<td>JHotDraw</td>
<td>AbstractCommand.seteEventDispatcher(), AbstractTool.checkUsable(), UndoableTool.createEventDispatcher(), CommandMenu.addMenuItems(), HtmlTextAreaFigure.markImageDirty(), AbstractTool.isUsable(), AbstractFigure.invalidate(), FigureAttributeConstant.addConstant(), CollectionFactory.createList()</td>
</tr>
<tr>
<td>Telecom</td>
<td>AbstractSimulation.say(), AbstractSimulation.report(), Timer.getTime(), Timing.aspectOff(), Timing.getTime()</td>
</tr>
</tbody>
</table>

**Table 6** The comparison between our approach with the approach in Paper [8].

<table>
<thead>
<tr>
<th>Case</th>
<th>Our approach</th>
<th>Approach in Paper [8]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of different executed methods</td>
<td>PetStore 46</td>
<td>—</td>
</tr>
<tr>
<td>Telecom 213</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Number of candidate methods (candidate cross-cutting concerns)</td>
<td>PetStore 7</td>
<td>10</td>
</tr>
<tr>
<td>Telecom 30</td>
<td>40</td>
<td>—</td>
</tr>
<tr>
<td>Number of effective methods (effective cross-cutting concerns)</td>
<td>PetStore 5</td>
<td>6</td>
</tr>
<tr>
<td>Telecom 10</td>
<td>5</td>
<td>—</td>
</tr>
<tr>
<td>Precision</td>
<td>PetStore 57.1%</td>
<td>40.0%</td>
</tr>
<tr>
<td>Telecom 35.3%</td>
<td>25.0%</td>
<td>—</td>
</tr>
<tr>
<td>Recall</td>
<td>PetStore 100%</td>
<td>83.3%</td>
</tr>
<tr>
<td>Telecom 83.3%</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

Table 6 shows the comparison between the approach presented in this paper and that of paper [8] in the number of different executed methods, the number of candidate methods, the number of effective methods and also the precision and recall. As can be seen, the precision of the approach presented in this paper has been significantly improved over that in paper [8] with the given three cases.

How much cross-cutting concerns can be found depends on the coverage of test cases. If we are able to run
the test cases that cover all codes of a system, we will surely find all the cross-cutting concerns. However, to the best of our knowledge, no researchers have demonstrated the recall of their approaches to the identification of cross-cutting concerns. The reason for this is that it is very difficult to know all the cross-cutting concerns that exist, especially in a very large system. In order to develop a rough understanding about the recall of our approach for a specific test case, we counted the total number of cross-cutting concerns in Telecom manually and found, using the chosen test case, that we could identify approximately 83.3% of total cross-cutting concerns.

Due to loops, polymorphism and fan-in aggregation, some methods do have anomalously large numbers of fan-ins. Such methods cannot be regarded as the cross-cutting concerns. Meanwhile, the method with a small number of fan-ins but suitable LCD and RCD indicates its low value to be a cross-cutting concern while refactoring codes. On the other hand, a small threshold of cross-cutting increases the number of methods to be investigated, whereas a big one may miss some real cross-cuttings.

We investigated how the different thresholds influenced the precision of identification. Figure 6 shows the changes in the number of identified candidate methods depending on the changes of the fan-in threshold \( T_{\text{fan-in}} \), whereas Fig. 7 shows the changes in the number of identified candidate methods depending on the changes of cross-cutting threshold \( T_{\text{cross-cutting}} \). As they show, the number of identified candidate methods reduces along with the increasing number of the fan-in thresholds or cross-cutting thresholds. In order to reduce the number of unconcerned methods and remain the concerned ones, we suggest setting the fan-in threshold \( T_{\text{fan-in}} \) to 5 and the cross-cutting threshold \( T_{\text{cross-cutting}} \) to 2 for JHotdrow, Petstore and Telecom.

7. Related Works

Aspect mining tries to identify cross-cutting concerns in existing systems to tackle the issues of code scattering and tangling. Currently, there are many approaches presented for aspect mining that can be generally divided into two kinds of approaches, i.e., static code analysis and dynamic code analysis, depending on whether running the programs or not. Besides, the combination of these two approaches is also frequently employed.

Tourwe and Mens put forward the notion of static formal concept analysis (FCA) to mine aspectual views [5]. They define an aspectual view as a set of source code entities that are structurally related in some way. These entities can be any source code artefact, such as a class hierarchy, a class, a method, a method parameter or an instance variable. Andrian and Andrey use an information retrieval approach to concept location in source code [17]. They map concepts expressed in natural language by the programmer to the relevant parts of the source code by using the approach of latent semantic indexing (LSI). Another static analysis approach includes [18]. All these approaches depend on good naming conventions when generating concepts or an LSI space. However, in practice different programmers in the software development process usually define the variable name according to their own preferences. Moreover, they may contain a lot of noise caused by the recognition results, thereby increasing the manual analysis of the workload.

Breu and Krinke proposed the first dynamic recognition approach to extract cross-cutting concerns on legacy systems based on analysing the methods’ execution traces [8]. They use program traces that are generated in different program executions as an underlying data pool. These traces are then investigated for recurring execution patterns based on different constraints, such as the requirement that the patterns have to exist in different calling contexts in the program trace. Although the concepts of execution relations presented in our paper are somewhat similar as those from [8], there are some differences. We ignore the neighbouring methods that are separated by statements because it is really very hard to determine the exact weaving places during code refactoring. In addition, if two methods in the same statement run in sequence, we also treat them as the exit-entry and entry-exit relations. Marin and Deursen identify cross-cutting concerns by measuring the fan-in [9]. They identify candidate aspects based on determining methods that are called from many different places (and, hence, have a high
fan-in) in a number of open-source Java systems. Our previous approach focuses on recurring execution patterns [19]. However, it depends heavily on manual processing after capturing the execution patterns. Moreover, although it introduces the recurring constraint that the cross-cutting concerns should satisfy, the recurring constraint itself cannot be quantified and is thus impossible to measure. Another dynamic analysis approach can be found in paper [20]. However, in these approaches, the called methods are all analysed regardless of the situation, whereby some of them are clearly incapable of being cross-cutting concerns. Consequently, the results may contain some noise, which greatly reduces the precision.

Ceccato and Tonella put forward a combined approach of concept analysis and traces analysis [21]. Execution traces are generated for those use cases that exercise the main functionalities of a given application. The relationship between execution traces and executed computational units (class methods) is subjected to concept analysis. Zhang and Guo, in [22], propose an automated approach called ‘clustering-based fan-in analysis’ which identifies aspect candidates together as groups. It is said that the approach can improve the efficiency of aspect mining and provide better support for refactoring. Some other combined approaches include [23], [24], etc.

Unlike the above approaches, the approach presented in this paper identifies the candidate cross-cutting concerns based on recurring execution relations and fan-in analysis. It measures the fan-in while running the test case to collect execution traces. Meanwhile, it filters out such methods as JDK-native ones that cannot be the candidate concerns. Afterwards, it collects all the methods with high fan-ins, which are frequently entered or quit under the same circumstances, as the candidate cross-cutting concerns. Our approach has no constraint on the coding and naming conventions of legacy systems as compared with static identification. Moreover, it can identify the potential candidate methods by analysing both methods’ execution relations and fan-ins to improve the precision when compared with the traditional trace analysis approach.

8. Conclusions

We present in this paper a novel approach to the identification of cross-cutting concerns based on recurring execution relations and the method’s fan-in. We present four types of execution relations, namely exit-entry, entry-exit, entry-entry and exit-exit, by considering the sequence of the methods’ invocation. We check the methods’ left cross-cutting degrees and right crossing degrees. The former ensures that the candidate recurs in a similar context, whereas the latter indicates how many times the candidate cross-cuts different methods. The final candidates are obtained from the high fan-in methods, which not only cross-cut others more times than a predefined threshold, but are always entered or quit under the similar context. In this way, we exclude some candidate cross-cutting concerns that are difficult to be refactored into aspects.

The fan-in analysis without considering execution traces may report lots of high fan-in methods due to many reasons, such as loops, polymorphism and fan-in aggregation, which greatly reduces the prediction accuracy. On the other hand, the analysis of only execution traces but not fan-ins may find that some candidate methods are not frequently invoked. The combination of these two methods increases the precision and is not straightforward. We ignore the methods with anomalously large number of fan-ins, and consider other methods with a relatively large number of fan-in and suitable LCD and RCD as the candidate cross-cutting concerns while analysing the captured execution traces.

Our approach is applicable when the software needs to be refactored based on the paradigm of aspect-oriented software development. The prerequisite of such refactoring is that we know the underlying cross-cutting concerns (methods) which need to be separated from the business concerns. In other words, we should mine the cross-cutting concerns (methods) before we refactor the codes according to the aspect-oriented paradigm. It is not necessary for our approach to change the codes or add stubs in order to mine the candidate cross-cutting concerns during the runtime. What we shall do is to merely write an AspectJ file and change the software to an AspectJ project, so as to identify the execution traces during the runtime. Therefore, it is not intrusive to the source codes and thus is very applicable for mining the candidate cross-cutting concerns (methods).

The effectiveness of our approach depends heavily on the test case chosen. The more codes the test case covers, the more cross-cutting concerns that might be identified. However, in the present case we chose a typical scenario based on our own experience as the test case that covered as many methods as possible. We are planning to study a heuristic approach to find the most effective execution traces (or their combination) in order to identify all the cross-cutting concerns. In addition, since the identification of cross-cutting concerns is a semi-automatic process [25], we will develop a complete IDE plug-in that can identify and extract the candidate cross-cutting concerns in a human-computer interaction environment. Last, but not least, the issue of how to automatically refactor the identified cross-cutting concerns into the aspects is also on the list of our future work.

Acknowledgments

The work is supported by China National Natural Science Foundation (No.60903053), Zhejiang Natural Science Foundation (No.LY12F02003), and the open project foundation of Zhejiang Provincial Key Laboratory of Network Technology and Information Security. The authors would also like to thank anonymous reviewers who made valuable suggestions to improve the quality of the paper.
References