ASPECT MINING ON REAL LEGACY CODE, APPLYING THREE DYNAMIC ANALYSIS TECHNIQUES

M. FERNANDA CAMPOS

Thesis submitted to the Office of Research and Graduate Studies in partial fulfillment of the requirements for the degree of Master of Science in Engineering

Advisor: YADRAN ETEROVIC S.

Santiago de Chile, Agosto 2011

© MMXI, MARÍA FERNANDA CAMPOS
© MMXI, MARÍA FERNANDA CAMPOS

Se autoriza la reproducción total o parcial, con fines académicos, por cualquier medio o procedimiento, incluyendo la cita bibliográfica que acredita al trabajo y a su autor.
ASPECT MINING ON REAL LEGACY CODE, APPLYING THREE DYNAMIC ANALYSIS TECHNIQUES

M. FERNANDA CAMPOS

Members of the Committee:
YADRAN ETEROVIC S.
ROSA ALARCÓN C.
STÉREN CHABERT
ANDRÉS GUESALAGA M.

Thesis submitted to the Office of Research and Graduate Studies in partial fulfillment of the requirements for the degree of Master of Science in Engineering

Santiago de Chile, Agosto 2011

© MMXI, MARÍA FERNANDA CAMPOS
Para mi papá, para las tres Marías
que iluminaron su vida, y para JP
que ilumina la mía.
ACKNOWLEDGEMENTS

First, I would like to extend my heartfelt gratitude to Professor Yadran Eterovic, for his encouragement, help and friendship along the process of this work. Our coffee meetings will be missed.

I would also like to thank the professors and staff at the department of computer science, for their teachings and help, for providing me with a second place to call home.

Thanks to my friends and family, to those present and those already gone; you have made me who I am, your support has made everything possible.

I would specially like to thank my mother, sister, and my boyfriend. Their unconditional love and support throughout this years has encouraged me to finish this work.

Finally, thank you dad; to where you are, I miss you. I made it.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acknowledgements</td>
<td>v</td>
</tr>
<tr>
<td>List of Tables</td>
<td>viii</td>
</tr>
<tr>
<td>List of Figures</td>
<td>ix</td>
</tr>
<tr>
<td>Abstract</td>
<td>x</td>
</tr>
<tr>
<td>Resumen</td>
<td>xi</td>
</tr>
<tr>
<td>Chapter 1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1. Static Analysis Techniques</td>
<td>1</td>
</tr>
<tr>
<td>1.2. Dynamic Analysis Techniques</td>
<td>2</td>
</tr>
<tr>
<td>1.3. Motivation and Objectives</td>
<td>2</td>
</tr>
<tr>
<td>Chapter 2. The dynamic analysis techniques</td>
<td>4</td>
</tr>
<tr>
<td>2.1. Description of the three techniques</td>
<td>4</td>
</tr>
<tr>
<td>2.1.1. Event traces</td>
<td>4</td>
</tr>
<tr>
<td>2.1.2. Formal concept analysis</td>
<td>5</td>
</tr>
<tr>
<td>2.1.3. Random walks</td>
<td>7</td>
</tr>
<tr>
<td>2.2. Applying the techniques in a small example</td>
<td>8</td>
</tr>
<tr>
<td>Chapter 3. Application to a real life program</td>
<td>13</td>
</tr>
<tr>
<td>3.1. Tools</td>
<td>13</td>
</tr>
<tr>
<td>3.2. The traces</td>
<td>15</td>
</tr>
<tr>
<td>Chapter 4. Results</td>
<td>17</td>
</tr>
<tr>
<td>4.1. Event traces</td>
<td>17</td>
</tr>
<tr>
<td>4.2. Random Walks</td>
<td>20</td>
</tr>
<tr>
<td>4.3. Concept Analysis</td>
<td>21</td>
</tr>
<tr>
<td>4.4. Combining the results</td>
<td>24</td>
</tr>
</tbody>
</table>
## List of Tables

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Trace size</td>
</tr>
<tr>
<td>3.1</td>
<td>Example of an execution trace file and the code that would generate it</td>
</tr>
<tr>
<td>3.2</td>
<td>Trace size</td>
</tr>
<tr>
<td>4.1</td>
<td>Execution relations results</td>
</tr>
<tr>
<td>4.2</td>
<td>Homogenous \textit{crosscutting concerns}</td>
</tr>
<tr>
<td>4.3</td>
<td>Homogenous \textit{crosscutting concerns} results for Random Walks algorithm</td>
</tr>
<tr>
<td>4.4</td>
<td>Concept lattice objects are the execution trace associated with specific use cases</td>
</tr>
<tr>
<td>4.5</td>
<td>Number of \textit{crosscutting concerns} found by concept analysis for an object group</td>
</tr>
<tr>
<td>4.6</td>
<td>Number of \textit{crosscutting concerns} found by each technique</td>
</tr>
</tbody>
</table>
List of Figures

2.1 Example trace ................................................................. 5

3.1 An execution trace tree in the visualization tool .................... 14

4.1 Additional concerns obtained by applying the technique to different traces from heart scans .............................................. 18

4.2 Additional concerns obtained by applying the technique to different traces from brain scans ........................................... 19

4.3 Additional concerns obtained by applying the technique to different sequences ...................................................... 20

4.4 Additional concerns obtained by applying the technique to different sequences ...................................................... 23

4.5 Additional concerns obtained by applying the technique to all sequences ................................................................. 24

4.6 Additional concerns obtained by applying the three techniques instead of one ............................................................. 25
Program comprehension is a practical field in software engineering concerned with understanding existing code for reuse, maintenance, refactoring and migration, with applications in many disciplines. In particular research in magnetic resonance imaging (MRI), requires reprogramming the software that controls the scanner, which in terms requires identifying the functions and parameters that must be changed.

Unfortunately, this particular software is very large, complex, non object oriented, almost not modularized and poorly documented. Therefore performing the changes and making sure that they are correct usually takes a very long time.

We propose that by applying aspect mining techniques to the program that controls a scanner we can obtain information about the properties of the program. We also propose that by applying a combination of techniques, we can improve the information we get.

We have chosen three aspect mining techniques, because they give us different information about the system. We have implemented the techniques to analyze execution traces, and we have obtained traces for two different type of sequences, a heart scan and a brain scan.

Results show information regarding the programs execution. They let us know what functions are related to the base of the program, like validations and parameters initialization, and which are related to a sequence type. We have also found out where some functionalities perform in the code.

Overall the information obtained regarding the programs execution can be very valuable to improve the current documentation.

Keywords: Aspects, Aspect mining, Execution relations, Random walks, Concept analysis, Legacy code
RESUMEN

La comprensión de programas es un área de la ingeniería de software dedicada a entender códigos existentes para la reutilización, mantención, reestructuración y migración, con aplicación en muchas disciplinas. En particular la investigación en imagenología por resonancia magnética (IRM), requiere reprogramar el programa que controla el resonador, lo que significa identificar las funciones y parámetros que deben modificarse.

Lamentablemente, este es un programa muy grande, complejo, no orientado a objetos, mal modularizado y pobremente documentado. Es por esto que realizar los cambios y probar que son correctos normalmente toma mucho tiempo.

Proponemos que mediante la aplicación de técnicas de minería de aspectos sobre el programa que controla un resonador podemos obtener información sobre las propiedades del programa. También proponemos que con la combinación de técnicas podemos mejorar la información que obtenemos.

Hemos escogido tres técnicas de minería de aspectos, por que nos entregarán distinta información del programa. Hemos implementado las técnicas de manera que estas analicen trazas de la ejecución del programa. Hemos obtenido trazas para dos tipos de secuencias diferentes: de corazón y de cerebro.

Los resultados nos entregan información sobre la ejecución del programa. Nos permiten saber que funciones pertenecen a la base del programa, como validaciones e inicialización de parámetros, y cuales están relacionadas con el tipo de secuencia. También hemos encontrado donde ciertas funcionalidades se ejecutan dentro del código.

Finalmente la información obtenida sobre la ejecución del programa es valiosa para mejorar la documentación actual.

**Palabras Claves:** Aspectos, Minería de aspectos, Código heredado
Chapter 1. INTRODUCTION

Aspect mining is a reverse engineering process, which consists in finding crosscutting concerns hidden in existing non aspect oriented software (Marin, Deursen, & Moonen, 2004). It is a well defined and actively studied problem, which is still far from being solved (Zhang & Jacobsen, 2007).

There are several aspect mining techniques: aspect identification through use case analysis (Rago, Abait, Marcos, & Diaz-Pace, 2009), static code analysis (Hannemann & Kiczales, 2001), some dynamic analysis techniques have been proposed (Breu & Krinke, 2004); (Tonella & Ceccato, 2004); (Safyallah & Sartipi, 2006). In (Nora, Said, & Fadila, 2006), a classification of these techniques has been proposed based on the characteristics of the crosscutting concerns the techniques are able to find.

Proposed techniques of aspect mining, similarly to aspect oriented programming, have been tested on small scale experiments (Baldi, Lopes, Linstead, & Bajracharya, 2008) and mainly on object oriented languages (Adams, Van Rompaey, Gibbs, & Coady, 2008). In this work we are concerned with the application of these techniques to a real life program, studying how they behave when applied to non object oriented legacy code, where these techniques may help understand the program’s organization and important features.

1.1. Static Analysis Techniques

Static analysis techniques are basically the search of patterns representing possible crosscutting concerns throughout the source code. Obtaining aspects using type-based and text based analysis has been proposed (Hannemann & Kiczales, 2001). Text based analysis searches for patterns in the names of methods and variables; it works better when strict naming conventions have been applied while writing the code. Type-based analysis has been proposed as a complement to that technique; it tries to find possible hidden concerns by studying the usage of types.
1.2. Dynamic Analysis Techniques

Several dynamic analysis techniques have been proposed for aspect mining. These techniques consist mainly of execution trace analysis: studying the program behavior through its execution.

(Breu & Krinke, 2004) analyze event traces by finding execution relations in the way the methods are called throughout a trace; aspect candidates are those relations which are fulfilled throughout the whole trace. (Tonella & Ceccato, 2004), propose using formal concept analysis on a group of traces to obtain possible crosscutting concerns. The use of a data mining technique for obtaining aspect by finding frequently used patterns, is proposed by (Safyallah & Sartipi, 2006).

Few comparisons between aspect mining techniques have been done so far (Cojocar, 2007). (Ceccato et al., 2006) selected 3 mixed techniques to compare and combine and have applied them to a benchmark program. Like Ceccato et al, we combine three techniques but focus on the application of these techniques to a real, non object oriented, legacy program, where these techniques may prove useful in obtaining hints of scattered code. All techniques have been implemented to use execution traces to obtain aspect candidates.

1.3. Motivation and Objectives

Research in magnetic resonance imaging (MRI) requires researchers to change the code that controls a scanner, so as to obtain non-standard behavior from the scanner. This task is difficult because of the complexity of the software. It involves making several small changes scattered throughout the code, and usually requires specialized training (Campos et al., 2009). Today, the task of programming a sequence can take months: understanding the scanner code and finding the places where the code must be changed is a time consuming task. If we obtain a better documentation of the scanner code and identify the parts of the code that are connected to the programming of sequences, we
can then help make it easier to program new MRI sequences and obtain faster results for research.

We believe that aspect mining can be a powerful tool in the search for the elements in the code that are related but nevertheless are scattered and, therefore, can be candidates for modularization. Because of its size and complexity and because few people really understand it, making any change to this code is very difficult. It is important to devise an approach for a better understanding of this code; finding possible crosscutting concerns may give us a way to go.

The application of aspect mining techniques to a non-object oriented code can demand a lot of work. Most techniques have been proposed and applied only to object oriented software, and rarely to real, non-benchmark programs. We want to show that they can also prove useful to other programs where refactoring is probably an even bigger necessity.

Our main hypothesis is that by applying aspect mining techniques to a real, large, and complex, non object-oriented program, we can obtain information about the properties of the program.

A second hypothesis us that by applying a combination of aspect mining techniques, we can improve the information we get.
Chapter 2. THE DYNAMIC ANALYSIS TECHNIQUES

We have chosen three aspect mining techniques, that give us different information about the system. The first, finds recurring execution traces, indicating general execution patterns in the code. The second technique looks for those functions from a same module that are present in all traces, i.e., scattered code, and also to core functionality of the code. It also can compute the list of functions that are only present in a certain trace, i.e., what functions are more related to a specific trace type. The third, identifies the most popular and less significant functions of the trace; it can also point out functions that crosscut the system. Since to obtain the crosscutting concerns we need to calculate the popularity and significance of each function in the trace, we can see this will also give us the list of most significant functions.

2.1. Description of the three techniques

2.1.1. Event traces

This technique was proposed by (Breu & Krinke, 2004). It is based on searching for specific types of execution relations that occur always throughout an execution trace. These are outside before, outside after, inside first and inside last. Outside before refers to a relation where for an execution of a function \( a \) the function \( b \) is executed completely just before \( a \). The reversed relation is called outside after, that is given an execution of \( a \), then \( b \) is executed immediately after \( a \) has finished. The inside first relation refers to when if there is an execution of function \( a \), then the first function called inside is \( b \). Similarly inside last relation is where if \( a \) is executed, then the last function called inside it is \( b \).

Two properties are defined over theses relations. First, a relation is called uniform if it exists always in the same composition. A relation is crosscutting, when it occurs in more than one calling context in the trace. The set of execution relations that comply with being both uniform and crosscutting are called aspect candidates.
The *uniform* constraint ensures that the execution traces are recurring patterns. The *crosscutting* constraint gives us all relations that occur in more than one context, and therefore can be considered to be scattered. This tells us that the *aspect candidates* that we will find with this technique are pairs of functions called always in the same order in different places of the program.

![Figure 2.1. Example trace](image)

As an example, suppose we have the execution trace shown in figure 2.1. First we must obtain all four types of execution relations:

- **Outside-Before:** \{ (g,h), (a,b), (b,a) \}
- **Outside-After:** \{ (h,g), (b,a), (a,b) \}
- **Inside-First:** \{ (b,root), (c,b), (g,c) \}
- **Inside-Last:** \{ (c,b), (h,c), (g,c) \}

Now we compute the aspect candidates, relations that are both uniform and crosscutting:

- **Uniform:** \{ (g,h), (a,b), (b,root), (c,b) \}
- **Crosscutting:** \{ (g,h), (c,b), (a,b) \}

### 2.1.2. Formal concept analysis

Formal concept analysis is a method of data analysis that provides a way of identifying maximal groupings of objects that have common attributes.Applying formal concept analysis to execution traces was proposed by [Tonella & Ceccato (2004)](Tonella & Ceccato (2004)). They propose
obtaining a concept lattice from the traces, where objects are the traces associated with several use cases and the executed functions in each trace are the attributes. A concept is the object plus its attributes. Applying concept analysis allows us to obtain the maximal grouping of methods (attributes) associated with certain scenarios or use cases (objects).

Aspect candidates are obtained when different functions from a same module are executed for more than one concept or when a concept has functions from different modules, the latter not being sufficient to identify a crosscutting concern. Scattered code can be found if for more than one of the traces we analyze we find calls to the same functions. This will help us understand the code, because it points out to those functions that are always executed in traces, i.e., the parts that don’t vary from one trace to another and are more likely, in charge of user interface, validations and initializations and not so much of the sequence it self.

Let’s say we have three traces, associated with use case a, use case b and use case c. Now each trace calls the following functions:

a: (vowel, letter).
b: (consonant, letter).
c: (consonant, letter).

From these we have a group of traces = {a, b, c} and a group of functions {vowel, consonant, letter}. Then we have the following concepts:

({b, c}, {consonant})
({a}, {vowel, letter})
({b, c}, {consonant, letter})
({a, b, c}, {letter})

Depending on the modules vowel, consonant and letter belong to they may or may not be aspect candidates. For example, if they all belong to the same module then all three functions would be candidates, since they are present in more than one concept.
2.1.3. Random walks

We have modified the technique proposed by (Zhang & Jacobsen [2007]), for finding crosscutting concerns through random walks, to apply it to execution traces. This technique is basically the application of a page rank algorithm to identify possible concerns by determining which elements are more popular and significant. We modified this proposed technique, and we calculate popularity and significance over execution traces instead of random walks. Popularity is the number of times an element is visited from different elements. Significance is the number of distinct elements an element visits. These ranks help us obtain potential crosscutting concerns in two ways:

*Homogeneous crosscutting:* popularity is higher than significance by a certain threshold. We will focus on these.

*Heterogeneous crosscutting:* elements with much higher significance than popularity.

By applying this technique we find the functions that are called most by different functions throughout the program execution, but which themselves don’t call many functions. High popularity of a function may hint at scattering in the code, low significance tells us that the function is probably not complex. This technique will also help us find those functions that are more complex and call many other functions; they are more likely to be core elements of the program.

Using again the trace in figure 2.1 if we calculate the popularity and significance of each element of the trace, we obtain the results shown in table 2.1

<table>
<thead>
<tr>
<th>Function</th>
<th>Popularity</th>
<th>Significance</th>
</tr>
</thead>
<tbody>
<tr>
<td>root</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>a</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>c</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>g</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>h</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Since all functions are called by the same function every time, then they are not very popular. Function \( c \) is the most significant because it calls more functions. Our candidates would be functions \( a, g \) and \( h \), which are the most popular and least significant.

### 2.2. Applying the techniques in a small example

All three techniques find possible concerns by searching for scattered methods in different ways. Even though the intersection of the results may not be empty, each technique is likely to add candidates.

Using event traces, we obtain concerns that are relations between methods. The results are candidates that repeat always in the same way through one trace. They are not necessarily the most called functions in the trace, neither are they necessarily called in more than one trace.

Concept analysis funds methods (attributes) that belong to one module and are called by more than one use case, or trace, i.e., methods that are recurring from one trace to another. Again, these methods are not necessarily the most called in each trace.

Finally popularity gives us candidates that are called most in a trace. These may not be in all traces, though are very likely to be in more than one.

Let’s take as an example the following code:

```c
int main( int menu, int a, int b )
if (is_valid_menu(menu))
    if (menu == 1)
        return sum (a, b)
    else if (menu == 2)
        return multiply(a,b)
    else
        return sum_and_double(a,b)
return 0

int is_valid_menu(int menu)
    if (menu == 1 || menu == 2 || menu == 3)
        return 1
    else
```

```c
8
```
return 0

int is_zero (int a)
    if (a == 0)
        return 1
    else
        return 0

int sum (int a, int b)
    return (a+b)

int sum_and_double (int a, int b)
    int c = sum(a,b)
    return multiply(c,2)

int multiply (a, b)
    if (is_zero(a) || is_zero(b))
        return 0
    int c = a
    for (int i=1; i<b; i++)
        c = sum(c,a)
    return c

If we consider that all executions start by calling the main function, then the execution of this code gives us the following trace when called with menu=1, a=2, b=3:

in:main
in:is_valid_menu
out:is_valid_menu
in:sum
out:sum
out:main

The following trace is obtained when main is called with menu=2, a=2, b=3:

in:main
in:is_valid_menu
out:is_valid_menu
in:multiply
in:is_zero
out:is_zero
in:sum
out:sum
The following trace is obtained when main is called with menu=3, a=2, b=3:

```
in:main
in:is_valid_menu
out:is_valid_menu
in:sum_and_double
in:sum
out:sum
in:multiply
in:is_zero
out:is_zero
in:sum
out:sum
out:multiply
out:sum_and_double
out:main
```

The crosscutting concerns found by the execution traces technique are pairs of functions that are always called in a certain order. For example:

- `is_valid_menu` is always called inside first of function `main`.
- `is_zero` is always called inside first of function `multiply`.
- `sum` is always inside first of function `sum_and_double`.
- `multiply` is always inside last of `sum_and_double`.

This technique also determines us which parts of the code execute for a given trace. For example, looking at trace with menu=1 we find that:

- `is_valid_menu` is always called outside before `sum`

But if we take a trace with menu=2 we find that:

- `sum` is outside after of function `is_zero`.
So concerns found in one trace are not necessarily found in another.

This information lets us know how the functions are called inside the program. It gives us an idea of the order in which they are called, and how this order varies when parameters change the trace type.

Concept analysis identifies *concerns* that are functions found in all the traces. For this particular case, all traces call *main, is_valid_menu* and *sum*.

We can also search by type of traces, i.e., only traces of type *menu=3* call *sum_and_double*, and traces of types *menu=2* and *menu=3* call *multiply* and *is_zero*.

Thus by applying this technique we learn which functions are specific to one type of trace, and which are more general and are always called.

The popularity technique determines, for traces of type *menu=2*, the following *cross-cutting concerns*: *is_zero* and *sum*.

But these functions are not called in all traces; for instance, in traces of type *menu=1* function *is_zero* is not called, and *sum* is only called once. This tells us which *concerns* are part of a certain functionality.

From this technique we learn which are the most called functions for a given trace. It identifies general functions that are called in all types of traces, and also to functions that are specific to certain types of traces.

Even though many of the *crosscutting concerns* may not turnout to be *aspects* they give us insight into the program and its composition. As the example in this section shows, we learn about the execution order, which functions are specific to certain traces, and which functions are global or called for all traces; thus, we begin to understand the program in a general sense.

Because the list of the scanner program aspects is not known, we will not evaluate the quality of the results as aspects. To validate the implementation of the techniques we have evaluated them over a benchmark program called JHotDraw. This program has a known list of crosscutting concerns, because it has been searched for aspects many times: [Marin]
et al., 2004), (Marin, Moonen, & Deursen, 2006), (Cojocar, 2007), (Yuen & Robillard, 2007), (Zhang & Jacobsen, 2007) and (Krinke & Breu, 2004). We took the aspect candidates obtained from applying these technique and compared them to those found in the bechmark program. Results obtained are equivalent to those described by (Breu & Krinke, 2004), (Tonella & Ceccato, 2004) and (Zhang & Jacobsen, 2007).

Since our goal is to understand the program, and not search for aspects, we will from now on refer to the results of applying the techniques on a non object-oriented code as crosscutting concerns.
Chapter 3. APPLICATION TO A REAL LIFE PROGRAM

Our focus is the application of the techniques, described in section 2, to a real life legacy program, in particular the one used to control a Philips scanner. The software that controls the scanner is large and complex: these computer programs were developed several years ago and have been modified many times. The program contains over 300,000 lines of code written in GOAL-C, a special-purpose programming language. It deals with properties represented by more than 6,000 global objects that appear in multiple files. A typical execution trace obtained by executing a real pulse sequence in the scanner may contain more than 30,000 function calls (Campos et al. 2009).

The program functions are organized into hundreds of files, which are grouped into components by separating them into a few folders. This organization of the code is neither intuitive nor clear. The current modularization makes it difficult to understand what functionality each component encapsulates or each file is concerned with. There is a division into few folders, files are divided into those concerning the parameter definitions (PDF), measurement program (MPF), scan geometry and more global modules.

We are applying techniques designed for object oriented programs, to do this, we treat each file as a module or class and the functions of the file as the class methods.

The purpose of applying aspect mining techniques to the scanner code is to identify the program’s concerns and to be able to understand them. This information may allow us to produce a better code documentation, since real encapsulation and modularization is unlikely, and may prove difficult or impossible due to ownership issues.

3.1. Tools

To obtain execution traces from the scanner program we have implemented a GOAL-C parser to insert print statements at the beginning and end of each function.

We obtain the traces by executing sequences in the scanner simulator. These traces indicate when the program enters a function and when it comes out. Table 3.1 shows what
a trace file looks like and a code fragment that can generate such a trace. The code is
divided into three files, with some functions in each. The trace shows when the execution
enters a function \( a \) in a file \( \text{file\_name\_k} \), with a line \( \text{file\_name\_k:in:a} \); and shows when
the execution exits the function, with a line like \( \text{file\_name\_k:out:a} \).

<table>
<thead>
<tr>
<th>file_name_1:in:root</th>
<th>file_name_1</th>
<th>file_name_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>file_name_1:in:b</td>
<td>function root() { b(); d(); }</td>
<td>function c { g(); h(); }</td>
</tr>
<tr>
<td>file_name_2:in:c</td>
<td>function b() { c(); }</td>
<td>function h() {}</td>
</tr>
<tr>
<td>file_name_1:in:g</td>
<td>function g() {}</td>
<td>function d() { e(); }</td>
</tr>
<tr>
<td>file_name_1:out:g</td>
<td></td>
<td></td>
</tr>
<tr>
<td>file_name_2:in:h</td>
<td></td>
<td></td>
</tr>
<tr>
<td>file_name_2:out:h</td>
<td></td>
<td></td>
</tr>
<tr>
<td>file_name_1:out:b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>file_name_2:out:c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>file_name_1:out:e</td>
<td></td>
<td></td>
</tr>
<tr>
<td>file_name_3:in:d</td>
<td></td>
<td></td>
</tr>
<tr>
<td>file_name_1:in:e</td>
<td></td>
<td></td>
</tr>
<tr>
<td>file_name_1:out:e</td>
<td></td>
<td></td>
</tr>
<tr>
<td>file_name_3:out:d</td>
<td></td>
<td></td>
</tr>
<tr>
<td>file_name_1:out:root</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

We have implemented a program for visualizing these traces (Campos et al., 2009).
Figure 3.1 shows the above trace file as seen in the visualization tool. The trace is dis-
played as tree of function calls, providing us a global view of the trace. The tool also
lets us search for functions within the trace, obtain the list of functions that are called and
determine the difference between two traces, among other things.

![Figure 3.1](image-url)
3.2. The traces

We used different traces, obtained by executing the scanner simulator. The size of each trace (number of lines and number of function calls) is shown in table 3.2. The name of a trace indicates the sequence type. For example, all traces that start with 614 are heart scans and those starting with 24_13 are brain scans. The scanner menu shows the 614 selection like \textit{3D\_BTFE\_NAV} and the brain scan selection with the name \textit{DW\_SSH}.

\begin{table}[h]
\centering
\caption{Trace size}
\begin{tabular}{|c|c|c|}
\hline
Trace & \# of lines & \# of functions \\
\hline
614 & 419728 & 1172 \\
614\_T2prepNO & 423364 & 1175 \\
614\_tfe10 & 641388 & 1173 \\
614\_tfe10\_halfscanYES & 437150 & 1174 \\
614\_tfe15 & 496596 & 1173 \\
614\_tfe25 & 380212 & 1173 \\
614\_triggerDelay200mms & 430736 & 1174 \\
614\_triggerDelay300mms & 427070 & 1174 \\
24\_13 & 41246 & 1128 \\
24\_13\_HalfScanNO\_TE130 & 57482 & 1137 \\
24\_13\_HalfScanNO\_TE400 & 55756 & 1137 \\
24\_13\_HalfScanNO\_TE400\_Slices44 & 69252 & 1140 \\
24\_13\_HalfScanNO\_TE130\_Slices11 & 70692 & 1141 \\
24\_13\_Slices11 & 50480 & 1132 \\
24\_13\_Slices44 & 54888 & 1132 \\
24\_13\_TE130 & 50694 & 1132 \\
24\_13\_TE400 & 48966 & 1132 \\
\hline
\end{tabular}
\end{table}

Trace names also indicate the particular parameter settings of each execution; for example:

\begin{itemize}
\item \textit{614.trace}: keeping the default values every parameter.
\item \textit{614\_tfe10.trace}: modifying the \textit{tfe} (turbo field echo) value to 10.
\item \textit{614\_tfe10\_halfscanYES.trace}: modifying the \textit{tfe} value to 10 and turning on \textit{halfscan}.
\item \textit{24\_13}: keeping default values.
\item \textit{24\_13\_Slices11}: setting the value of \textit{slices} to 11.
\item \textit{24\_13\_TE130.trace}: setting the \textit{TE} (echo time) to 130.
\end{itemize}
We can see from table 3.2 that even though the trace size (in number of lines) vary between traces, the number of different functions called remains invariable. For example, let’s look at traces 24_13_TE130 and 24_13_TE400: the number of functions called remains the same even though we modified the *echo time*; this is most likely because modifying the parameter value doesn’t add new functionality to the trace, so no new functions are called. We looked closer at these traces and verified that the functions executed in both cases are in fact the same. Similar cases can be appreciated if we look at traces: 614_tfe1, 614_tfe15 and 614_tfe25; 24_13_Slices11 and 24_13_Slices44; 24_13_HalfScanNO_TE130 and 24_13_HalfScanNO_TE400; 614_triggerDelay200mms and 614_triggerDelay300mms.
Chapter 4. RESULTS

We now analyze the results obtained by applying the three aspect mining techniques to our set of real traces from the execution of the Phillips scanner code. First, we present the results of each technique separately.

4.1. Event traces

From each trace we compute the set of relations outside before, outside after, inside first and inside last. We then apply the constraints uniform and crosscutting, to the resulting relations. The execution relations that are both uniform and crosscutting become our crosscutting concerns. Table 4.1 shows the results.

<table>
<thead>
<tr>
<th>Trace Name</th>
<th># of Exec. rel.</th>
<th># of Uniform &amp; crosscutting</th>
</tr>
</thead>
<tbody>
<tr>
<td>614</td>
<td>5755</td>
<td>312</td>
</tr>
<tr>
<td>614_T2prepNO</td>
<td>5779</td>
<td>315</td>
</tr>
<tr>
<td>614_tfe10</td>
<td>5765</td>
<td>315</td>
</tr>
<tr>
<td>614_tfe10_halfscanYES</td>
<td>5767</td>
<td>313</td>
</tr>
<tr>
<td>614_tfe15</td>
<td>5765</td>
<td>315</td>
</tr>
<tr>
<td>614_tfe25</td>
<td>5765</td>
<td>315</td>
</tr>
<tr>
<td>614_triggerDelay200mms</td>
<td>5767</td>
<td>313</td>
</tr>
<tr>
<td>614_triggerDelay300mms</td>
<td>5767</td>
<td>313</td>
</tr>
<tr>
<td>24_13</td>
<td>5466</td>
<td>275</td>
</tr>
<tr>
<td>24_13_HalfScanNO_TE130</td>
<td>5568</td>
<td>278</td>
</tr>
<tr>
<td>24_13_HalfScanNO_TE400</td>
<td>5568</td>
<td>277</td>
</tr>
<tr>
<td>24_13_HalfScanNO_TE400_Slices44</td>
<td>5587</td>
<td>278</td>
</tr>
<tr>
<td>24_13_HalfScanNO_TE130_Slices11</td>
<td>5591</td>
<td>279</td>
</tr>
<tr>
<td>24_13_Slices11</td>
<td>5495</td>
<td>279</td>
</tr>
<tr>
<td>24_13_Slices44</td>
<td>5495</td>
<td>278</td>
</tr>
<tr>
<td>24_13_TE130</td>
<td>5500</td>
<td>278</td>
</tr>
<tr>
<td>24_13_TE400</td>
<td>5500</td>
<td>278</td>
</tr>
</tbody>
</table>

First let’s analyze some groups of traces. Traces 614_tfe10, 614_tfe15 and 614_tfe25 not only have the same number of crosscutting concerns, but the crosscutting concerns for these three traces are exactly the same. This indicates that the value of the turbo field echo (tfe) doesn’t affect the order of the functions called along the trace. The same is true for
the value of the trigger delay parameter in traces 614_triggerDelay200mms and 614_triggerDelay300mms; and for the echo time in traces 24_13_TE130 and 24_13_TE400.

On the other hand, turning on certain parameters, such as halfscan (a technique used to reduce scan time), does affect the results; i.e., trace 614_tfe10 has two more crosscutting concerns than trace 614_tfe10_halfscanYES: we obtained two more concerns when halfscan is turned off.

From each of the heart sequences we obtained between 312 and 315 crosscutting concerns. Since these sequences have much in common the crosscutting concerns repeat from one sequence to another. Finally, we obtained a total of 317 crosscutting concerns.
When we perform the same exercise with the traces from brain sequences, these give us between 275 and 279 concerns each. Discarding the repeated results we get a total of 281 different concerns.

If we look for uniform and crosscutting execution relationships in the different types of heart sequences we obtain between 2 and 5 more concerns; similarly, by applying this technique to different types of brain traces we get between 2 and 6 more concerns. The numbers are shown in figures 4.1 and 4.2.

More interestingly, by combining the concerns from both type of traces we obtain a total of 364 different concerns, that is 49 more than if we had used only traces 614_tfe15 or 614_tfe25; and 89 more than if we had used the trace 24_13 which has the least number

---

**Figure 4.2.** Additional concerns obtained by applying the technique to different traces from brain scans
of results. The combination produces an increment between 15% and 32%, as shown in figure 4.3. Thus we identify more concerns by applying the aspect mining technique to different sequences instead of to the same sequence with different parameters. Different sequences are more likely to produce execution traces with more diverse function calls (unlike traces from a same sequences type that vary parameter values).

4.2. Random Walks

The random walks aspect mining technique computes popularity and significance for each of the traces; the results are shown in table 4.2. Crosscutting concerns are the functions that have been evaluated with high popularity and low significance. We defined a simple threshold to classify the functions to be considered crosscutting concerns. The method looks for crosscutting concerns which are homogeneously scattered throughout the program, i.e., are called from many different places, making them popular.
TABLE 4.2. Homogenous crosscutting concerns

<table>
<thead>
<tr>
<th>Trace Name</th>
<th># Homogeneous CC</th>
</tr>
</thead>
<tbody>
<tr>
<td>614</td>
<td>11</td>
</tr>
<tr>
<td>614_T2prepNO</td>
<td>11</td>
</tr>
<tr>
<td>614_tfe10</td>
<td>11</td>
</tr>
<tr>
<td>614_tfe10_halfscanYES</td>
<td>11</td>
</tr>
<tr>
<td>614_tfe15</td>
<td>11</td>
</tr>
<tr>
<td>614_tfe25</td>
<td>11</td>
</tr>
<tr>
<td>614_triggerDelay200mms</td>
<td>11</td>
</tr>
<tr>
<td>614_triggerDelay300mms</td>
<td>11</td>
</tr>
<tr>
<td>24_13</td>
<td>12</td>
</tr>
<tr>
<td>24_13_HalfScanNO_TE130</td>
<td>12</td>
</tr>
<tr>
<td>24_13_HalfScanNO_TE400</td>
<td>12</td>
</tr>
<tr>
<td>24_13_HalfScanNO_TE400_Slices44</td>
<td>12</td>
</tr>
<tr>
<td>24_13_HalfScanNO_TE130_Slices11</td>
<td>12</td>
</tr>
<tr>
<td>24_13_Slices11</td>
<td>12</td>
</tr>
<tr>
<td>24_13_Slices44</td>
<td>12</td>
</tr>
<tr>
<td>24_13_TE130</td>
<td>12</td>
</tr>
<tr>
<td>24_13_TE400</td>
<td>12</td>
</tr>
</tbody>
</table>

We obtained the same homogenous concerns for all traces that come from sequences of the type: 11 homogenous crosscutting concerns for heart scan traces, and 12 for brain scans. This means that the concerns are not susceptible to the variations of the parameters in the trace of one sequence type, but are affected by the sequence type.

If we combine the results from both types of sequences we get a total of 13 different crosscutting concerns as shown in table 4.3.

Similar to the case of using event traces (although the numbers are different), we identify nine crosscutting concerns if we apply the random walks technique to two different types of traces. Though one or two concerns don’t seem a lot, in practice they represent a gain of between eight and 20 percent, as shown in figure 4.4.

4.3. Concept Analysis

To apply concept analysis we consider each trace as an object associated with a specific use case and the functions belonging to the trace as the attributes. Table 4.4 shows which objects are associated with each use case.
### Table 4.3. Homogenous crosscutting concerns results for Random Walks algorithm

<table>
<thead>
<tr>
<th>Homogeneous CC</th>
<th>CC from 614 traces</th>
<th>CC from 24_13 traces</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPGAC_sw_bit_not_present</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>MPG_get_pdfBehaviour</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>MPG_get_pdf_appl</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>MPGAC_sw_bit_present</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>MPGAC_sw_bit_not_present ... 2</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>MPGGEOM_matrix_to_sg_matrix</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>MPGPS_conv_pat_dir_to_ps</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>get_main_system_type</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>MMPGN_set_shims</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>MPIACQ_nus_method</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>MPGPS_conv_orient_to_ps</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>MPGAC_sw_bit_not_present ... 3</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>MPICOIL_get_app_sar_limit</td>
<td>yes</td>
<td>no</td>
</tr>
</tbody>
</table>

### Table 4.4. Concept lattice objects are the execution trace associated with specific use cases.

<table>
<thead>
<tr>
<th>Objects</th>
<th>Use case</th>
</tr>
</thead>
<tbody>
<tr>
<td>614</td>
<td>heart scan with default values</td>
</tr>
<tr>
<td>614_T2prepNO</td>
<td>heart scan with no T2 preparation (pulse used to increase blood contrast)</td>
</tr>
<tr>
<td>614_tfe10</td>
<td>heart scan with turbo field echo value 10</td>
</tr>
<tr>
<td>614_tfe10_halfscanYES</td>
<td>heart scan with turbo field echo 10 and halfscan on</td>
</tr>
<tr>
<td>614_tfe15</td>
<td>heart scan with turbo field echo value 15</td>
</tr>
<tr>
<td>614_tfe25</td>
<td>heart scan with turbo field echo value 25</td>
</tr>
<tr>
<td>614_triggerDelay200mms</td>
<td>heart scan with trigger delay of 200mms</td>
</tr>
<tr>
<td>614_triggerDelay300mms</td>
<td>heart scan with trigger delay of 300mms</td>
</tr>
<tr>
<td>24_13</td>
<td>brain scan with default values</td>
</tr>
<tr>
<td>24_13_HalfScanNO_TE130</td>
<td>brain scan with no halfscan and a echo time of 130</td>
</tr>
<tr>
<td>24_13_HalfScanNO_TE400</td>
<td>brain scan with no halfscan and a echo time of 400</td>
</tr>
<tr>
<td>24_13_HalfScanNO_TE400_Slices44</td>
<td>brain scan with no halfscan and echo time of 400 and 44 slices</td>
</tr>
<tr>
<td>24_13_HalfScanNO_TE130_Slices11</td>
<td>brain scan with no halfscan and echo time of 130 and 11 slices</td>
</tr>
<tr>
<td>24_13_Slices11</td>
<td>brain scan with 11 slices</td>
</tr>
<tr>
<td>24_13_Slices44</td>
<td>brain scan with 44 slices</td>
</tr>
<tr>
<td>24_13_TE130</td>
<td>brain scan with an echo time of 130</td>
</tr>
<tr>
<td>24_13_TE400</td>
<td>brain scan with an echo time of 400</td>
</tr>
</tbody>
</table>
A case of scattered code is found when the same function is found in more than one concept (see section 2.1.2). If the same function is found in more than one concept it is probably a scattered function. We show the results in Table 4.5.

**Table 4.5. Number of crosscutting concerns found by concept analysis for an object group**

<table>
<thead>
<tr>
<th>Objects</th>
<th>Crosscutting concerns</th>
</tr>
</thead>
<tbody>
<tr>
<td>614, 24_13</td>
<td>973</td>
</tr>
<tr>
<td>all 614 traces</td>
<td>1034</td>
</tr>
<tr>
<td>all 24_13 traces</td>
<td>1098</td>
</tr>
<tr>
<td>all traces</td>
<td>1258</td>
</tr>
</tbody>
</table>

First, we execute the concept analysis algorithm for the heart and brain scans with default values. We obtain 973 crosscutting concerns; these are the functions that have been called in both traces, and are more likely to be scattered.

Next, we apply concept analysis to all heart traces and find 1034 crosscutting concerns. This shows that most functions in the trace are repeated through the different traces,
therefore we find many concerns. The same is true when we apply the technique to all brain scans: we find 1098 concerns, showing that most functions in scans are repeated in more than one trace.

![Bar graph showing additional concerns obtained by applying the technique to all sequences](image)

Figure 4.5. Additional concerns obtained by applying the technique to all sequences

Finally, applying the technique over all the traces we obtained 1258 crosscutting concerns. Had we considered any other group of traces (other than the maximal) we would have fewer concerns, because each trace makes it more likely to have repeated functions. We obtained 311 more concerns than had we used only the default value traces, 250 more than using all the heart scans, and 186 more than if we had only used brain scans; this is shown in figure 4.5. These represent a 33, 25 and 17 percent more concerns respectively.

### 4.4. Combining the results

How much have we gained by applying the three techniques instead of applying just one? Each technique gave us 364, 13 and 1258 crosscutting concerns, respectively, as shown in table 4.6. By construction, those given by execution relations are different from
the rest. Thus, instead of finding just 364, or just 1258 from the concept analysis, we found 1622. We found that 8 of the concerns obtained from concept analysis are different from those from random walks, so in total we have 1630 crosscutting concerns.

Table 4.6. Number of crosscutting concerns found by each technique

<table>
<thead>
<tr>
<th>Technique</th>
<th>Crosscutting concerns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exec Relations</td>
<td>364</td>
</tr>
<tr>
<td>Random Walks</td>
<td>13</td>
</tr>
<tr>
<td>Concept Analysis</td>
<td>1258</td>
</tr>
<tr>
<td>Union of Techniques</td>
<td>1630</td>
</tr>
</tbody>
</table>

Figure 4.6. Additional concerns obtained by applying the three techniques instead of one.

The largest gain is with respect to having just applied the random walks algorithm. But even in the case where we gained the least number of crosscutting concerns the increment was a 29 percent.
Chapter 5. WHAT WE LEARNT ABOUT THE PROGRAM

We have, so far, analyzed the results in terms of number of crosscutting concerns obtained by each technique. Our main issue with evaluating the quality of the identified concerns is that we don’t know what aspects actually exist in the code. We cannot evaluate the results and analyze them using suggested measurements, such as how many turn out to be real crosscutting concerns, how many are mistakenly recommended, or how many known aspects are not found (Cojocar, 2007).

However, our purpose is to discover and understand (some of) the programs properties. For this we must take a closer look at the results and analyze what they represent in the code. Since we have over 1000 crosscutting concerns, in this section we analyze a few from each aspect mining technique to show that applying these techniques to a non object-oriented program provides useful information about the program.

5.1. Concerns from execution relations

From the 364 crosscutting concerns obtained from this technique 232 are present on both types of sequences; of the 132 remaining 85 are concerns that come from the heart sequences and 47 from the brain scans. This shows that certain relations apply only for certain sequences.

Looking at the code for some of the 232 common concerns we see that, as expected, most of them are core functions: those that have more to do with the scan execution than with the sequence itself.

We see multiple repetitions of some functions in the relations; in particular, two stand out: MPG_get_pdf_appl and MPG_get_pdf_behaviour. Not only are these functions called many times, but they are called from different modules, from at least 20 files, and from many functions. This suggests the code is coupled. Other functions present
similar modularization issues, such as `MPGAC_sw_bit_not_present` and `MPGAC_sw_bit_present`. If we look at these functions’ purposes, we see that they are very simple, and, as their names suggest, are only in charge of returning the value of a given option.

Multiple types of validations can be found in the common relations; many of them form inside first relations with `MPGAC_sw_bit_not_present` and `MPGAC_sw_bit_present`. There are at least 13 functions of the form `MODULE_val2_ranges`, from different modules calling these two functions. If we look at the code, we see these functions are in charge of setting ranges for the parameters in the module during a second validation phase: `val2`.

Another set of relations stands out: the inside last relations formed between various functions in (`idef_val1`, `idef_val2`, `idef_val3`, `idef12_val`) and function `idef_check_unexpected_conflicts`. If we look at these functions we see that they control validation phases 1, 2 and 3, and they always call last `idef_check_unexpected_conflicts`. This function is described as “checks for unexpected conflicts that are introduced by coding errors.”, so part of the validation is to make sure no conflicts exist.

Three relations between module `MPIUSEGEOM` validations and `MPG_get_pdf_behaviour`, where the latter is always called inside first, can be found: they are the functions that perform the validation phases, in charge of enabling and disabling certain parameters depending on the value of the `pdf_behaviour`.

In the common relations we also find relations like `MPIMA_set_softkey_screen` calls inside first `MPIEX_softkey_screen`. In the code, we see `MPIMA_set_softkey_screen` is a function called when a user selects a certain protocol page in the scan simulator; `MPIEX_softkey_screen` masks enabled parameters during validation. Both functions seem to be part of the user interface for simulation of scans.

Also part of the scan simulator we found function `MPIMN_validate_init`, that according to its description validates initial parameters, function `MPIMA_scan_init`, which prepares the scans control parameters for normal scan and function `MPIMN_validate_exit` which must be called in case user modifications are successful after a “proceed”.
This begins to clear up which parts belong to the Parameter Definition (PDF) part of the program: those that are concerned with the setup and validation of the parameters modifications. What is interesting is that most relations found in common are from this part of the program. Only, four relations were found from the measurement part of the program (MPF).

Looking at the relations found only for the heart scans we find an interesting group of relations that involve functions:

- icard8_decr_max_phases
- icard8_get_total_phases_dur
- icard8_max_phases
- MPICARD_8_opt_and_max_phases
- icard8_set_nr_phases_range

These functions are directly involved with heart phases calculations.

A few relations involving functions igeo_total_nr_slicesets are found in the brain traces, they are involved with the calculation of the slice sets for the scan.

We can also find in the brain scans a group of inside first relations between functions:

- idiff_calc_hom_ec_in_mps
- idiff_calc_lin_ec_area_in_mps
- idiff5_calc_nr_diff_dirs

and the function MPGCEO_m_matrix_to_sg_matrix. This last function is in charge of storing a matrix structure, that can be calculated in the function idiff_calc_lin_ec_area_-in_mps in a scan geometry type matrix.

5.2. Concerns from concept analysis

This technique gives us a list of functions that appear on all traces. This means that each candidate function is executed at least once in each trace. The fact that it repeats in
different executions may be pointing to a scattering issue, but also refers to functions that are part of the base of the program like user interface functions or certain validations.

We find in the aspect candidates functions related to SAR restrictions, this makes sense since they it is factor that should always be calculated before a scan. Initializations functions are also between these candidates: MMIPARS_init, MMIRFE_initialise, MMITRACK_initialise, among others.

Some functions that execute scans call our attention, taking a closer look we can see that: MMPAS_scan, performs the measurement of the automatic shiming; MMPPO_scan, performs power optimization; MMPPU_scan performs pickup optimization; MMPPW_scan, patient width preparation phase measurements and MMPTM_scan, described as "main procedure for tuning of all coils". All these functions are can be considered part of the core of these traces, since they take the measurements necessary for obtaining the sequences (MPF).

The function MPIMN_scan_exec is also part of these candidates, if we take a closer look at it we can see it is in charge of executing the PDF for the scan. It is called after the "start scan" button is pressed and before the scan starts. This function seems to be in charge of calling certain initializations and setting up some parameters for the scan.

If we take the results of applying the technique only to 614 traces and compare them with those obtained by applying the technique only to 24_13 traces, we can obtain the list of candidates that appear in one of the types of sequences and not in the other. This points toward functions that are executed for all 614 traces and don’t execute for 24_13, and vice versa.

Some functions in charge of scans are only called for one sequence type, for example MMIRFE_scan is only called for the brain scans, this functions performs the Radio frequency energy (RFE) scan measurements.

We can find a similar function for the heart scans, MMIFFE_scan, in charge of performing the Fast field echo (FFE) scan measurements. So the first function has something to do with this type of heart sequences (614), and the second with brain sequences of this
We have also learned that depending on the trace these functions may or not appear.

5.3. Concerns from random walks

Analyzing the results obtained for this technique we have found that some concerns are functions that already called our attention in the previous techniques. For example functions like:

- \texttt{MPGAC\_sw\_bit\_not\_present}
- \texttt{MPGAC\_sw\_bit\_present}
- \texttt{MPG\_get\_pdf\_behaviour}
- \texttt{MPG\_get\_pdf\_appl}

These are not only part in crosscutting and uniform relations but are also part of the most popular and least significant.

Only one concern is unique for each type of trace. For the hearts scans \texttt{MPICOIL\_get\_app\_sar\_limit}, a function that gets the maximum amount of SAR (specific absorption rate) allowed. Candidate function \texttt{MPGEO\_m\_matrix\_to\_sg\_matrix}, that stores a matrix structure in a scan geometry type matrix, was also a concern from the execution relations.

All candidates for traces of a sequence type are the same, and even between the traces most of them repeat. Since they don’t vary with the sequence, and are called multiple times, they can be considered part of the applications core. Because this technique requires the candidates to have low significance, the results are very simple functions that are very much like get and set functions.

5.4. Concerns obtained in general

Even though the functions identified are obviously in the code, it is not easy to spot them just by looking at the code, which comprises hundreds of files, and over 300,000 lines of code. Besides it's not easy to determine which is the purpose of each function.
Our analysis allows us to do exactly this: to identify relevant functions and to determine their purpose.

The functions identified are those that get executed during a trace; the program has functions that are not really executed, because they have been replace but not removed. It is not easy to spot those functions by simply looking at the code.

So let's take a closer look at the obtained concerns, and what they let us know about the program.

In terms of the execution order of the program we learn things like:

- Minimal repetition time is always calculated by function `MPIFFE_8_min_rep_time` before calling `idef_calc_rel_sar`.
- Scan control parameters are always prepared before obtaining the the patient ref_id (`MPIMA_scan_init` is called outside before `MPA_cur_patient_ref_id`).
- Initial parameters are always validated before deciding if the given geometry should be applied (`MPIMN_validate_init` is called outside before `MPG_set_apply_geometry`)
- `MPIMN_validate` is always called at the end of `MPIMN_validate_init`.
- Pdf behavior (`MPG_get_pdf_behaviour`) is always called at the end of `MPIMN_scan_exec`.

These relations are only a few of a list of over 200 concerns that tell us more about the order of execution of functions within the program for all traces.

We can also obtain concerns that are specific for a type of trace. For instance, in the heart sequences:

- `icard8_decr_max_phases` is called outside before `icard8_get_total_phases_dur`:
  Maximum number of heart phases is calculated before calculating the total phases duration.
MPICARD_8_phase_dur is called inside first of icard8_get_total_phases_dur: the first function called during the calculation of the phase duration is a function that calculates the total phases duration.

Again, these are a few examples out of a list of 85 concerns found for the hear sequences and 47 for brain traces.

The results from concept analysis give us an extensive list of concerns, functions that are called in each of the traces. These functions are most likely to be related to the general parts of the execution of the program, like parameter setup and validation.

We found functions concerned with parameter setup. An example of these is function iacq0_echo_times, called during validation phase 0. We also found a group of functions in charge of enabling and setup of EX parameters, in validation phase 1, these are:

\[ \begin{align*}
  \text{iacq1\_B0\_map} \\
  \text{iacq1\_coca} \\
  \text{iacq1\_echo\_times} \\
  \text{iacq1\_epi} \\
  \text{iacq1\_foldover\_and\_averaging} \\
  \text{iacq1\_misc, iacq1\_rep\_times} \\
  \text{iacq1\_scan\_matrix} \\
  \text{iacq1\_scan\_technique} \\
  \text{iacq1\_shot\_mode} \\
  \text{iacq1\_water\_fat\_shift}
\end{align*} \]

Setup functions from validation phase 2 and 3 were also found.

Also part of all traces we found a group of local functions concerned with updating the reconstruction parameters, like:

\[ \begin{align*}
  \text{irc\_update\_acq\_mode\_pars} \\
  \text{irc\_update\_B0\_map\_pars} \\
  \text{ irc\_update\_ctag\_pars}
\end{align*} \]
irc\_update\_for\_shutters

We found a number of validation functions, like:

\texttt{MPIREPP\_valN}
\texttt{MPIRESP\_valN}
\texttt{MPIREST\_valN}

Here \texttt{valN} indicates that it executes validation phase \textit{N}. For instance, \texttt{MPIREPP\_val0}, \texttt{MPIREPP\_val1} and \texttt{MPIREPP\_val3} perform validation phases one, two and three respectively.

For functions corresponding to phase two we can also find functions of the type:

\texttt{MPIREPP\_val2\_comb}
\texttt{MPIREPP\_val2\_ranges}

The first checks combinations during the validation. The second sets ranges during the second phase of validation.

We also took a look at those functions that are common only for traces of one type.

For heart traces of type \texttt{614} we learned that local functions concerning \textit{fast field echo} (ffe) are called. These are functions like:

\texttt{iffe\_calc\_freq}
\texttt{iffe\_calc\_phase}
\texttt{iffe\_filter\_oversampling}
\texttt{iffe\_sq\_init\_epi}.

In 24\_13 traces of brain scans we found a group of local functions concerning \textit{refocused field echo} (RFE), or \textit{spin echo}. Some examples of these functions are:

\texttt{irfe\_sq3\_init\_epi}
\texttt{irfe3\_tse\_factor\_range}
\texttt{irfe5\_refocusing\_sweeps}
This is consistent with \textit{MMIFFE\_scan} being called only for heart traces, and \textit{MMIRFE\_scan} only for brain scans.

From the random walks technique we obtained a brief list of more general functions that are called for both types of traces. Such is the case of:

\begin{verbatim}
MPGAC\_sw\_bit\_not\_present
MPGAC\_sw\_bit\_present
\end{verbatim}

These are in charge of responding if an option (the \textit{sw\_bit}) is present.

Also called in all traces, we found:

\begin{verbatim}
MPG\_get\_pdf\_behaviour
MPG\_get\_pdf\_appl
\end{verbatim}

Both are concerned with returning values about the parameter definition program, these are called during validation phases.

Only two functions appear popular only one type of trace. Function \textit{MPICOIL\_get\_app\_sar\_limit} is popular for heart scans, but if we take a look at brain traces it is also called for those sequences, only not enough times for it to be popular. This tells us that for heart scans the maximum SAR is calculated more times than for the brain scans. Similarly with \textit{MPGGE0\_m\_matrix\_to\_sg\_matrix}, it is popular for brain scans but not for hearts scans.

These are only some examples of things we have learned from applying the techniques to the program traces. Concerns obtained for execution traces and concept analysis gave us much more information, random walks gave a us few more concerns. All this shows out hypothesis correct, we can learn important information to understand a program by applying aspect mining techniques.
Chapter 6. CONCLUSIONS AND FUTURE RESEARCH

6.1. Conclusion

We have selected and implemented three aspect mining techniques based on dynamic analysis. We have modified them to obtain candidates from execution traces. We have applied these techniques to a full sized, tangled and non object-oriented legacy code.

Applying these techniques to a larger number of execution traces, with different use case scenarios proves useful for obtaining better results even for each particular technique.

The combination of techniques has proven to be successful in finding more possible concerns in a real life code. A quantitative analysis of the aspect candidates has been made for the results obtained from the scanners traces.

These techniques have helped us to understand in a better way the program with its execution, modularization and phases. They let us obtain valuable information about the code that can help us improve the current documentation. For example, we now know that certain functions are associated with specific types of sequences: \textit{MMIFFE\_scan}, is in charge of performing the \textit{Fast field echo (FFE)} scan measurements for 614 heart sequences. We have found a certain group of functions performing SAR validations that execute for all sequences. Other validation functions have been found for the different phases of validation that are performed for scans.

Finally, our hypothesis about the use of aspect mining technique to gain knowledge of a legacy code has given the expected results. Applying the techniques has given us information of the code that we didn’t know how to obtain, because its size made it difficult to approach. With the obtained information we can now improve the current documentation.
6.2. Products

Most aspect mining tools are made to be applied over programs in java or other object oriented based languages, since the scanner code is in such a particular language we have had to implement our own tools for the job.

A parser for obtaining traces from the scanners program has been implemented, this allows us to print the beginning and end of each function so we know which modules of the program are being executed each time we run it.

We have also implemented a tool that allows us to view, search and expand a given trace. Allowing us to view it as an execution tree. With this tool we may also apply each of the three aspect mining techniques to any software if we can obtain from it execution traces with the given format.

6.3. Future research

We have still have many possible paths we can follow to continue our research from this point on. The application of this results to obtain a better modularization of the scanners code can be a way to go. We don’t necessarily aim at obtaining the aspects as such, but using these candidates can help to see where the scattered code is so we can refactor and encapsulate it, to obtain a more approachable and maintainable code.

For investigating this particular code and its aspects it could also help to focus on parts of the code which are of more interest to the scanner sequences by leaving out those branches concerned with the GUI and with the set up of the parameters of the sequence. This with the objective of finding aspects more related with sequences giving us a chance to help MRI sequence investigation.

Applying the techniques to a broader group of sequence types, so we can obtain information regarding the functions execute specifically for each case is a priority, because improving the documentation is still our main goal.
We could also try to find better ways of combining the three techniques. For instance, we could apply more traces to concept analysis or consider a more restrictive application of this algorithm so as to aim for less but more accurate results. We could aim to find a better selection of traces for the random walks algorithm given that results don’t vary much for different traces, we could try applying it to just default traces from different sequence types.
References


Hannemann, J., & Kiczales, G. (2001). Overcoming the prevalent decomposition of legacy code. In In workshop on advanced separation of concerns at the international


