Delft University of Technology
Software Engineering Research Group
Technical Report Series

Proceedings

First International Workshop

Towards Evaluation of Aspect Mining — TEAM 2006 —

July 4, 2006, Nantes, France
co-located with 20th European Conference on
Object-Oriented Programming (ECOOP 2006)

Organized by
Silvia Breu, Leon Moonen,
Magiel Bruntink and Jens Krinke

Report TUD-SERG-2006-012

TUD-SERG-2006-012

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
http://www.se.ewi.tudelft.nl/techreports/

For more information about the Software Engineering Research Group:
http://www.se.ewi.tudelft.nl/

Table of Contents

Introduction

Workshop Background

Aspect-oriented software development (AOSD) has emerged over the last decade as a paradigm for separation of concerns, especially those crosscutting concerns that were difficult to decompose and isolate with earlier programming methodologies and resulted in scattered and tangled implementation of functionality. As such, AOSD promises significant benefits in the areas of software comprehension, maintenance and evolution and could play an important role in the revitalization of existing (legacy) systems.

This promise has driven an increasing number of researchers to investigate the challenge of identifying crosscutting concerns in existing systems (also known as aspect mining) using various forms of program analysis. However, despite initial efforts to define a common benchmark, the community is still lacking an evaluative framework that can be used to compare and evaluate the quality of various approaches to aspect mining.

This workshop aims at developing such a framework, which consists amongst others of a clear classification of different types of crosscutting concerns (including a description of their distinguishing features) as well as well-defined methods to assess different analysis approaches with respect to established quality criteria such as precision, recall, scalability, usability, etc.

Motivation, Topics, and Goals

The motivation for the workshop lies in the emerging number of approaches to (semi-)automatically identify crosscutting concerns in existing software systems. As the community and approaches mature, the desire grows to systematically compare and assess the various approaches to identify strengths, weaknesses, commonalities and differences and discover open issues for future investigation.

The goal of this workshop is to advance the state-of-the-art in evaluation and comparison of aspect mining techniques. To achieve this goal, we bring together practitioners, researchers, academics, and students working in the area of aspect mining, and more generally in the area of aspect oriented software development, to share experiences, consolidate successful techniques, collect guidelines, and identify open issues for future work.

One of the first requirements for evaluation is a proper classification of kinds of crosscutting concerns and their distinguishing features. Without such classification, detection results can hardly be compared between various approaches. In addition, we need well-defined methods to objectively compare aspect mining techniques and assess their quality attributes such as precision, recall, scalability, etc. We aim to work towards two main results, the first being a prerequisite for the latter:

1. to establish a (more complete) classification of (kinds of) crosscutting concerns, including a description of their distinguishing features.

2. to develop an evaluative framework to assess and compare aspect mining techniques.

Organizers

Silvia Breu	University of Cambridge, UK
Leon Moonen	Delft University of Technology & CWI, The Netherlands
Magiel Bruntink	CWI, The Netherlands
Jens Krinke	Fernuniversität in Hagen, Germany

Program Committee

Elisa Baniassad	Chinese University of Hongkong, China
Silvia Breu	University of Cambridge, UK
Magiel Bruntink	CWI, The Netherlands
Yvonne Coady	University of Victoria, Canada
Jens Krinke	Fernuniversität in Hagen, Germany
Christian Lindig	Saarland University, Germany
Marius Marin	Delft University of Technology, The Netherlands
Kim Mens	Université catholique de Louvain, Belgium
Leon Moonen	Delft University of Technology & CWI, The Netherlands
Lori Pollock	University of Delaware, USA
Awais Rashid	Lancaster University, UK
Martin Robillard	McGill University, Canada
Paolo Tonella	ITC-irst, Italy
Tom Tourwé	CWI, The Netherlands

Technical Papers

Evaluating EA-Miner: Are Early Aspect Mining Techniques Effective?

Ruzanna Chitchyan
Computing Department
Lancaster University
Lancaster, LA1 4WA, UK
rouza@comp.lancs.ac.uk

Américo Sampaio
Computing Department
Lancaster University
Lancaster, LA1 4WA, UK
a.sampaio@comp.lancs.ac.uk

Awais Rashid, Paul Rayson
Computing Department
Lancaster University
Lancaster, LA1 4WA, UK
awais/paul @comp.lancs.ac.uk

ABSTRACT

Identifying and analyzing aspectual requirements manually is very resource intensive due to the broadly scoped nature of aspects and the large volumes and ambiguity of requirements elicitation data.

In this paper we present an evaluation of EA-Miner – a requirements-level aspect identification tool. The tool is evaluated by comparing the performance of an analyst using the tool with that of carrying out the same tasks manually. To obtain objective results, the evaluation is repeated for three case studies of various sizes. Our evaluation demonstrates the high degree of accuracy provided by EA-Miner and the considerable reduction in time and effort afforded by it.

Keywords

aspect-oriented requirements engineering, aspect mining, aspect identification, requirements elicitation, tools, evaluation.

1. INTRODUCTION

Aspect-Oriented Requirements Engineering (AORE) [1] [2] [3] [4] has emerged as a new way to modularize and reason about crosscutting concerns during requirements engineering. AORE extends the notion of separation of concerns in RE (e.g., viewpoints, use cases, goals, etc.) with that of requirements-level aspects. Such aspects modularize requirements that affect and constrain other requirements; examples of requirements-level aspects are: security, availability, distribution, etc. Requirements pertaining to these concerns are often (fully or partially) scattered in the statements of other requirements. By explicitly modularizing crosscutting concerns at the requirements-level, AORE makes it possible to reason about such concerns from early on in the software lifecycle.

Identification of aspectual requirements is, however, a non-trivial task. Firstly, as is the case for identifying relevant concerns using any RE technique, one often has to mine for aspects in large volumes of input documents. Documents, such as interview transcripts, are frequently imprecise, full of apparent contradictions and missing essential information. Secondly, parts of aspectual concerns can often be scattered across a document or even across documents making their identification difficult. This is further compounded by factors such as the occurrence of similar, often incomplete requirements in several places, mutual influence of requirements, difference of language (user vocabulary) used to express same or similar requirements and implicit requirement implication.

Undertaking such identification tasks manually is often very time-consuming and costly. For instance, for an analyst with an average reading speed, it would take 1.5-2 minutes (at the rate of 250-350 words per minute) to read a 500-word-long problem for comprehension. Identifying key concepts, concerns and crosscutting relationships requires even more time and effort. When we extrapolate this data to manual processing of large documents, the average personnel effort required is substantial.

Therefore, like other RE techniques, AORE requires effective and scalable tool support in order for its benefits to be fully exploited in analyzing large scale problems.

In this paper we evaluate the effectiveness of our EA-Miner tool that automates identification[1] of aspectual (i.e. crosscutting) and non-aspectual concerns (i.e. base concerns), as well as structuring of requirements according to the identified concerns. The tool employs Natural Language Processing (NLP) techniques and has been previously reported in detail in [1, 5, 6].

In this paper we do not propose any "new evaluation frameworks" for aspect mining, but instead demonstrate the merits of our aspect-mining approach and how we evaluate it.

In section 2 we present an overview of the tool for readers not familiar with it. In section 3 we evaluate the tool[2], based on our experiences with analysis of three problem descriptions of varying sizes. Section 4 presents some related work while section 5 concludes the paper.

2. EA-Miner

EA-Miner is layered on top of our existing WMATRIX suite of natural language tools [6, 7] to support the requirements engineer in identification of both aspectual and non-aspectual concerns [1, 5].

WMATRIX is a web-based collection of corpus-based NLP tools. It uses a combination of part-of-speech and semantic tagging, frequency analysis and concordances (i.e. words in context) to identify concepts of potential significance in a given text. WMATRIX assigns part-of-speech tags for each word in the text (98% precision), then uses semantic analysis to group related words and multi-word expressions into conceptual categories. If a word has several meanings and can be assigned to more than one semantic category, the analyses helps to identify the most likely category for each word in a given text, taking into account the context where the word occurs.

---

[1] As discussed in the related work section, several other tools for AORE exist, but all of them require initial manual input for concerns.

[2] This evaluation is briefly mentioned in [6]. This paper presents more detailed discussion.

EA-Miner utilizes the part-of-speech tagging and semantic analysis produced by WMATRIX and complements it with its own lexicon for crosscutting concerns and requirement document representation models (e.g., viewpoints model, use cases model, etc.).

For identification of non-functional crosscutting concerns (which are often strong candidates for aspects), the EA-Miner lexicon builds on top of (and extends) non-functional requirement trees of the NFR framework [8]. Non-functional aspects are identified by assigning semantically close words to the sub-groups for each NFR category, thus all NFR-related aspects are identified. Additionally, a lexicon for any new non-functional aspects that are not included into NFR, but are deemed to be useful can be incorporated into the tool lexicon.

For identification of functional crosscutting concerns, EA-Miner uses a Theme/Doc-like [2] strategy, detecting the repeated occurrences of action words, which may suggest presence of a functional aspect. For instance consider the scenario in Figure 1 which shows the EA-Miner output pertaining to the description of an online auction system. EA-Miner lexicon helps to identify the words *authorised* and *logging* as semantically related to the *security* concern and suggests *security* as a non-functional aspect. It also alerts the requirements engineer (not shown in Figure 1) that the *bid* functionality is mentioned in many requirements, which may indicate some crosscutting association of *bidding*.

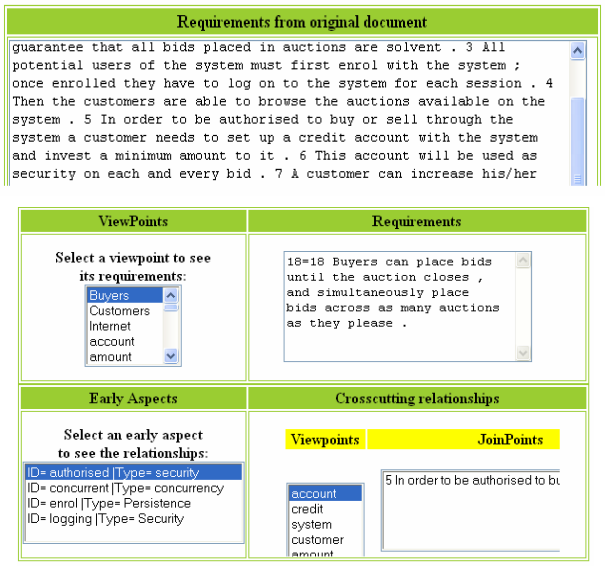

Figure 1: EA-Miner tool

EA-Miner also facilitates the production of the aspect model and requirements document representation with chosen structures. For instance (cf. Figure 1), EA-Miner helps to automatically identify viewpoints (e.g., buyers, customers, etc.), using the WMATRIX part-of-speech annotation – all nouns are identified as potential viewpoints. Since the list of nouns for large documents can be very long, EA-Miner applies several reduction strategies by:

1. using lemmatization to recognize words that have the same root (e.g. *customer* and *customers*) and treating these as one viewpoint;
2. using dictionaries of synonyms to amalgamate words with the same meaning (e.g. *client* and *customer*);
3. using WMATRIX frequency norms to consider as suggested viewpoints only significantly overused nouns.

EA-Miner then provides the list of requirements related to each viewpoint and its related aspects. Note that EA-Miner is not limited to viewpoint-based structuring of requirements. Other requirements models can be *plugged into* the tool. For instance, it can be applied to develop use cases style requirements documentation by identifying possible use cases from the action verbs (e.g., sell, buy, bid, etc.) and relating corresponding requirements to them. Similar reduction strategies to those outlined above are also usable in such instances.

3. EVALUATION OF EA-MINER

The most time consuming activities in AORE pertain to the identification of the crosscutting (aspects) and non-crosscutting (i.e., viewpoints in our experiment) concerns and to structuring of the requirements according to these concerns. Our evaluation of EA-Miner helps us to compare its performance with regards to these activities with a corresponding manual analysis.

3.1 The Evaluation Framework

We have compared results of EA-Miner with the manual analysis of three problem descriptions of varying sizes and document structures.

The first problem description used was the *auction system* described in [9]. This is a simple example, considering the size of the input document (443 words, 1 page). The second problem description was the light control system [10]. The size of this file is significantly larger than the previous example (3671 words, 11 pages). The third problem description was that of a library system used in Lancaster University. The document used as input to the tool is part of the requirements specification documentation of the system. The size of this file is larger than the previous two (6504 words, 29 pages).

The execution of both methods (tool-based and manual) was carried out independently by three different requirements engineers (two with similar level of expertise and one expert) in order to avoid biased results. We collected data on the *time required* in each case as well as data on the *quality of the output.*

To measure the *required time,* in the tool-based approach, the tool logged the time for concern identification while the requirements engineer used a chronometer to log the time for structuring the requirements into the concerns and screening out the impertinent ones. In the manual approach the requirements engineer used a chronometer to measure the time spent on the same set of activities.

The *quality* of the outcome was measured as the *number of correctly identified concerns*, and the number of *false positives*.

The number of correctly identified concerns (i.e., viewpoints and aspects) can alternatively be expressed as a *ratio (or percentage) of the correctly identified concerns* against the total number of correct concerns present in the problem description. The concerns were deemed *correctly identified* if they corresponded to those manually identified by an independent senior requirements engineer. The senior requirements engineer (who independently identified the relevant viewpoints and aspects) also normalized the results from the other two analysts by equating relevant abstractions where different names or granularities were used by the two participating engineers.

The metric for *false positive* abstractions (viewpoints and aspects) shows the number of concerns identified by the tool, but

considered wrongly identified by the senior requirements engineer. Similar to the number of correctly identified concerns, this metric can also be expressed as rate, i.e., the number of false positives/ total number of identified concerns.
It is evident, that in this work we have opted not to develop any dedicated aspectual evaluation framework, instead we rely on simple evaluation criteria of efficiency and quality that can clearly demonstrate the utility of our approach. We also believe that these simple criteria should act as starting point of evaluation for any requirements-level aspect mining approaches.

3.2 Results of Evaluation

As illustrated in Figure 2, the EA-Miner-based analysis consistently outperformed the manual one. In terms of *required time,* for the smaller auction system description it was approximately 20 times faster (4 minutes 45 seconds vs. 90 minutes). For the larger documents the performance of the tool was even better: approximately 125 and 120 times faster for the light control and library systems respectively.
It is important to note that in both the tool-based and manual analysis the largest part of time was expended not on reading and analysis, but on structuring the requirements per viewpoints and aspects. However, the time taken for structuring was much smaller for the tool-based approach as EA-Miner suggests the possible structuring of requirements into viewpoints and aspects.

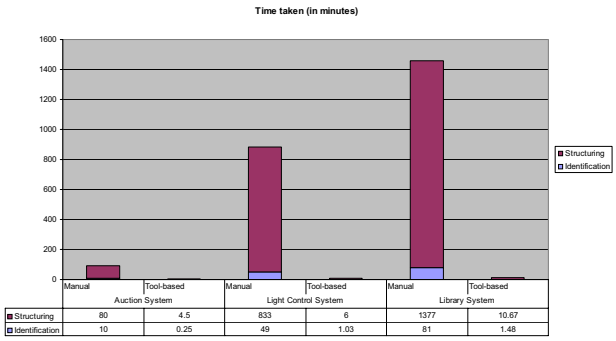

Figure 2: Comparison of Time Spent

In terms of *correctness of viewpoints* identified (Figure 3), the output of the two analyses was the same for the smaller (auction system) problem. However, the tool-based analysis was more accurate than the manual analysis for the two larger problem descriptions (with a 100% match with the senior engineer's analysis for the library system). This shows that the tool suite can significantly aid the requirements engineer in identifying a good set of viewpoints.

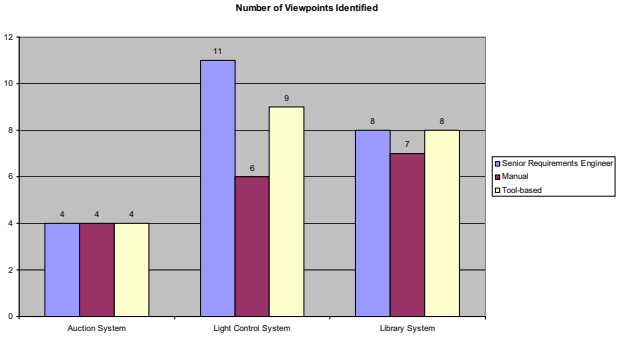

Figure 3: Comparison of Correct Viewpoints Identified

The comparison of the *number of aspects identified* by the two analyses is shown in Figure 4. Both sets of results are comparable for the light control system. For the auction system, the manual analysis missed one aspect (persistence).
Interesting results were observed for the library system where the manual analysis identified 7 out of the 8 aspects independently identified by the senior engineer, while the tool-based analysis identified only 4. This was due to the fact that vocabulary for such aspects as *standards and protocols* (mentioned in the library system) was not part of the EA-Miner lexicon. The lexicon is continuously refined and extended based on new vocabulary for aspects identified by the requirements engineers. Relevant vocabulary has, therefore, now been added to the lexicon. As more analyses are performed in other case studies, and the tool lexicon is populated with more aspect-related vocabulary, the performance of the tool on this criterion will improve further.

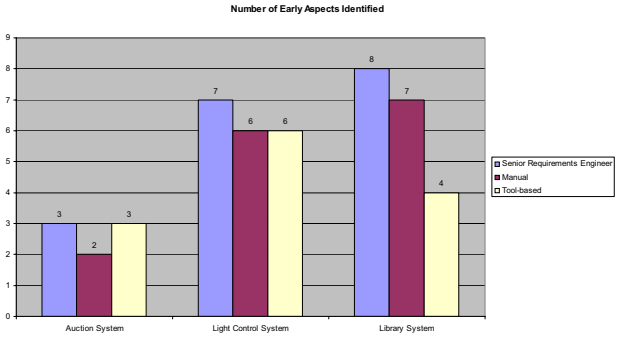

Figure 4: Comparison of Correct Aspects Identified

The differences in the two evaluation methods in terms of *false positives* became noticeable in evaluation of the two larger problem descriptions. The rate of false positives *for viewpoints* was approximately 0.1 for the tool suit and zero for the manual analysis. On the other hand, for the tool suite, the rate of false positive *aspects* was zero compared to 0.3 for the manual analyst. The subsequent interviews with the analysts helped explain these contrasts. For viewpoint identification, the tool considers all nouns as potential candidates. However, usually subjects in a sentence tend to make stronger candidates for the purpose. As viewpoints are well understood, the analyst working manually placed a stronger emphasis on subjects as viewpoints. Conversely, aspects at the requirements-level are not as well-understood abstractions. Thus, the manual analyst preferred to err on the side of caution identifying more aspects than needed. In contrast, the analyst working with the tools was more clearly guided with regards to potential aspects, hence resulting in no false positives.
Thus, in our experiment the tool proved to be useful and effective. In our view, this experiment demonstrates the value of the tool suite in helping with the adoption of aspect-oriented requirements engineering techniques, especially when extending existing viewpoints- or use case-based approaches with the notion of aspects.

4. RELATED WORK

Aspect identification at requirements level is still a research problem. Presently most work focuses on studying the crosscutting characteristics of known non-functional concerns. The only notable exception to this is the Theme/Doc [2] approach which allows to visually identify the crosscutting functional relationships between a set of (manually supplied) pre-defined

concerns in requirements. While EA-Miner was initially inspired by Theme/Doc idea, we based our approach on corpus-based NLP work and automation, thus addressing the issues of efficiency and scalability.

Another approach that uses NLP for aspect identification is presented in [11]. This work is motivated by a view that crosscutting concerns in OO code are often caused due to scattering of *actions*. NLP is used to identify such *actions* in method names and method related comments. These *actions* then can be viewed as a separate virtual concern to reduce program navigation and search costs.

Code level aspect mining is also addressed in [12][13]. [12] focuses on studying the relationships between the execution traces of the functionalities and the computational units (class methods) invoked during each execution. Crosscutting is suggested when features are made up by methods that belong to more than one class and each class contributes to more than one feature.

The Aspect Mining Tool (AMT) [13] is based on a combination of text based (pattern matching) and type-based search to identify aspect candidates in source code. The tool helps to provide visualizations showing in which classes a given pattern and/or type occurs.

5. CONCLUSION

In order to facilitate a wide adoption of aspect-oriented software development, it is necessary to offset the complexity and effort involved in aspect identification. Aspects manifest themselves first in the initial system requirements. Therefore, this is the point where we must address the associated complexity and effort by means of scalable tools that have a high degree of accuracy. Once the aspects have been identified, they can be traced, through refinements, into the solution space, i.e. the architecture, design and implementation. Therefore, automation of aspect identification and structuring is an area of key importance in aspect-oriented requirements engineering.

Our tool, EA-Miner, is aimed at providing effective and scalable support for concern identification and structuring. While we are still working on further development and improvement of the tool our first evaluation, presented in this paper, shows significant results in terms of time-efficiency and output quality for the AORE process.

We do not propose any new evaluation framework as first and foremost aspect mining tools and techniques must demonstrate their effectiveness in comparison with non-aspect-oriented or manual approaches. In our view, this is the first stepping stone towards a more extensive evaluation framework – such comparative studies are critical for Aspect-Oriented Software Development. They will also provide an adequate means of comparison of different AO approaches against each other, when these approaches peruse the same goals (e.g. identification of aspects in requirements, etc.).

A second step for such a framework should be the incorporation of existing work on metrics designed to evaluate aspect-oriented modularity, e.g., [14-16]. Such metrics not only provide a testbed to compare different approaches but also help analyze the results with regards to relevant quality criteria such as coupling, cohesion and separation of concerns.

6. ACKNOWLEDGMENTS

This work is supported by European Commission grant IST-2-004349: European Network of Excellence on Aspect-Oriented Software Development (AOSD-Europe), 2004-2008.

7. REFERENCES

[1] A. Sampaio et. al, "EA-Miner: a Tool for Automating Aspect-Oriented Requirements Identification," Automated Software Engineering (ASE 2005), Long Beach, California, USA, 2005.

[2] E. Baniassad and S. Clarke, "Theme: An Approach for Aspect-Oriented Analysis and Design," Int'l Conference on Software Engineering, Edinburgh, Scotland, UK, 2004.

[3] V. Ambriola and V. Gervasi, "Processing natural language requirements," presented at International Conference on Automated Software Engineering, Los Alamitos, 1997.

[4] I. Sommerville et. al, "Viewpoints for requirements elicitation: a practical approach," Int'l Conf. of Software Eng. (ICRE'98), Colorado Springs, Colorado, USA, 1998.

[5] A. Sampaio et. al, "Mining Aspects in Requirements," Workshop on Early Aspects held with AOSD 2005, Chicago, Illinois, USA, 2005.

[6] R. Chitchyan et. al, "A Tool Suit for Aspect-Oriented Requirements Engineering," Workshop on Early Aspects (held at ICSE 2006), Shanghai, China, 2006.

[7] P. Rayson, WMATRIX software, Lancaster University, URL: http://www.comp.lancs.ac.uk/ucrel/wmatrix/, 2005.

[8] L. Chung, B. A. Nixon, E. Yu, and J. Mylopoulos, *Non-Functional Requirements in Software Engineering*: Kluwer Academic Publishers, 2000.

[9] Web Site: *Auction System Problem Description*, http://lgl.epfl.ch/research/fondue/case-studies/auction/problem-description.html, Ecole Polytechnique Fédérale de Lausanne (EPFL), Lausanne, Switzerland, 2005.

[10] G. University of Kaiserslautern, "Web Site: The Light Control Case Study: Problem Description."

[11] D. Shepherd et. al, "Towards Supporting On-Demand Virtual Remodularization Using Program Graphs," Aspect -Oriented Software Development, Bonn, Germany, 2006.

[12] P. Tonella and M. Ceccato, "Aspect Mining through the Formal Concept Analysis of Execution Traces," 11th Working Conference on Reverse Engineering (WCRE'04), Delft University of Technology, the Netherlands, 2004.

[13] J. Hannemann and G. Kiczales, "Overcoming the Prevalent Decomposition in Legacy Code," in *Workshop on Advanced Separation of Concerns in Software Engineering (ICSE 2001)*, 2001.

[14] A. Garcia et. al, "Modularizing Design Patterns with Aspects: A Quantitative Study," Aspect-Oriented Software Development (AOSD'05), Chicago, USA, 2005.

[15] N. Cacho et. al, "Composing Design Patterns: A Scalability Study of Aspect-Oriented Programming," Aspect-Oriented Software Development, Bonn, Germany, 2006.

[16] A. F. Garcia et. al, "Modularizing Design Patterns with Aspects: A Quantitative Study," *Transactions on Aspect-Oriented Software Development*, pp. 36-74, 2006.

Automatic Mining Of Context Passing In Java Programs

Linda M. Seiter
John Carroll University
20700 North Park Boulevard
University Heights, OH USA
011-216-397-1948
lseiter@jcu.edu

ABSTRACT

Context-dependent computing is becoming increasingly necessary within the growing fields of distributed, service-oriented, ubiquitous, and autonomic computing. Context awareness involves the ability of a program to customize its behavior based on the context in which it is executing. This presents a challenge in statically scoped languages like Java, as it may require information available in one scope to be made available at another. In this paper, we investigate several AOP solutions that have been proposed for addressing different notions of context-dependent computation. We then discuss a potential framework for assessing tools that might be developed for mining and visualizing evidence of context-dependent computation within Java programs, the goal being the automatic refactoring into AOP solutions. Finally, we present an Eclipse plugin that has been developed to automatically mine and visualize context-dependent control flow in the form of parameter passing within Java programs.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Contructs and Features – *abstract data types, polymorphism, control structures*

General Terms

Design, Languages, Verification.

1. INTRODUCTION

Before discussing how to evaluate and compare aspect mining tools, we first present an overview of several AOP techniques have been proposed for modularizing the cross-cutting concern of context-aware computation. Each attempts to encapsulate a different notion of context-based behavior composition. One of the simplest notions of context-dependent computation involves calling-context. For example, a method may decide how to proceed based on information about the object that invoked the method (i.e. the calling context). In a statically scoped language like Java, the calling context is not available within the scope of the called method, thus context passing is traditionally achieved by the propagation of the caller's identity as a parameter along sequences of method invocations. This often results in API pollution as well as loss of modularity. Figure 1 shows an example in which the Service class implements its functionality by delegating to a Worker object, (possibly by routing through an intermediary object). The Worker object requires information concerning the object that requests the service in order to decide how to perform its work. The Caller context is passed as a parameter from *Service.doService* to *Worker.doWork*. There exist several non-AOP solutions to context passing, including techniques that provide reflective access to call histories [9], programming language support for dynamic context [10], dynamic variables [14], as well as declarative mechanisms for defining transportation graphs [5].

```
public class Caller {
    public void requestService(Service service) {
       service.doService(this,…);
    }
}
public class Service {
     Worker worker;
     public void doService(Object context, …) {
          worker.doWork (context, …);
     }
}
public class Worker {
   public void doWork(Object context,…) {
        //test context to determine behavior
        if (context. …….) …..
  }
}
```

FIGURE 1: Context passing through method parameter

Coady et al [1] proposed an initial AOP solution to context passing that used the *cflow* primitive to capture the calling context. A similar technique has been suggested by Laddad, which he refers to as the *wormhole design pattern* [4]. A wormhole involves the declaration of three pointcuts: (1) a source pointcut *callercontext* on the calling method of a client/service class, (2) a target pointcut *workercontext* on the called method of the worker class that requires access to the calling context, and (3) a *wormhole* pointcut that uses cflow to restrict the call path to joinpoints matched by the source pointcut that occur within the control flow of the target pointcut. Figure 2 contains a template of the wormhole pattern.

```
public aspect ContextPassing {
 pointcut  callercontext(Caller c) :
              this (c ) && call (void Service.doService(…) ;
 pointcut  workercontext(Worker w):
              target(w) && call(void Worker.doWork(…);
 pointcut  wormhole(Caller c, Worker w):
              cflow(callercontext(c)  && workercontext(w));
 around(Caller c, Worker w):
             wormhole(c,w) {   advice body     }
}
```

FIGURE 2: Wormhole Template

The use of the *cflow* pointcut expression restricts the context information that is available at runtime to the current contents of the call stack. In particular, the most recent client context encountered on the call stack is available at the target joinpoint. Cottenier and Elrad propose *contextual pointcut expressions* [2] as a mechanism for collecting information about the context in which a target joinpoint is executed, including context information about all client contexts that have been encountered along the call path, not just the most recent. This allows more accessibility to the life-cycle of a joinpoint, rather than simply its most recent occurrence. They implement a visitor that accumulates information about each dynamic joinpoint that matches a pointcut. Thus, path-specific customizations may be made using a greater historical record of context.

Herzeel et al propose temporal-based context awareness using HALO, a pointcut language designed to encapsulate context-awareness across an application's lifetime [15]. HALO is built upon an underlying system that snapshots context state throughout an application based on triggers defined by temporal pointcuts. The pointcut mechanism allows reference to past join points and the context in which they occurred, including reference to time intervals.

The *dflow* pointcut proposed by Masuhara and Kawauchi serves to trigger advice based on whether a value of an object at a target joinpoint originates from a value that was computed at some source joinpoint preceding it on the call stack [7]. The *dflow* pointcut relies on the history of where a value originates along the call stack.

Context-Aware Aspects proposed by Tanter et al [12] takes a unique approach of separating the definition of a context from its use within an aspect. They propose a framework to support context definitions as objects which are stateful, composable and potentially parameterized. A context-aware aspect is an aspect whose behavior depends on the context, perhaps in defining a pointcut or when applying advice. The framework supports contexts that may exist over time, thus it does not limit aspects to access only the most recent contexts.

2. ASSESSING ASPECT MINING TECHNIQUES

We summarize these varying AOP approaches to modularizing the context passing concern based on their pointcut definitions:

- cflow –recent client joinpoint context available to target joinpoint
- contextual, temporal – historical record of client joinpoint contexts available to target joinpoint
- dflow – historical record of dataflow along call stack available at target joinpoint
- context aware – better separation of concern between context and aspect that uses it

An existing Java program may contain numerous occurrences of the different types of context-dependent computation that each of these techniques propose to modularize. The goal of an aspect-mining tool would be to detect such occurrences, present them in a useful manner to a programmer, and support the eventual refactoring of such occurrences into an AOP solution. We propose several questions to be used in assessing the value of a particular mining approach.

- How do we detect context-aware computation?
- How do we visually represent context-aware computation?
- Can we support automatic refactoring to AOP?

A given technique for mining context-awareness must define and then detect the following:

- Context perception: what contexts are of interest? where are they *defined* (source), where are they *used* (target)?
- Context acquisition: what is the existing program mechanism for enabling source contexts to be *available* at targets?
- Context-dependent computation: *how* is the source context used at the target?

Once an occurrence of a context-aware computation is found in an existing program, the question arises as to whether the code can or should be refactored into an AOP solution. If so, can the refactoring be automated or is manual intervention required. Visual depiction of the context-dependent computation, for example depiction of a path or flow along which the context is passed, along with potential refactoring implications, is extremely valuable in facilitating program evolution.

3. AN ECLIPSE PLUGIN FOR MINING CONTEXT PASSING

We have developed an Eclipse plug-in called *ContextFlowMiner* that searches Java programs for parameter passing over a sequence of method calls. Presently the plugin focuses on patterns of context passing that could be refactored into the wormhole design pattern. It detects the propagation of an object along a sequence of method calls as a parameter. In this section, we briefly discuss some of the issues involved in mining such context passing flows in order to support the programmer toward proper refactoring to an AOP solution.

The primary goal of the *ContextFlowMiner* plugin is to search through the Java files within an Eclipse project in order to detect occurrences of parameter passing $m_i \rightarrow n_j$, in which an object that is passed into a method ***m*** as the i^{th} parameter is subsequently passed as the ^{jth} parameter to method *n* that is called within the

method body of ***m***. The plugin detects both general parameter passing paths (object passed into a method through any variable reference) as well as context passing paths (object passed into method through "*this*" variable reference). A summarized listing of all detected parameter passing paths is presented to the programmer; the programmer may then select a parameter passing path to view graphically.

The ultimate benefit of using such a tool would be to facilitate the programmer in making an informed decision as to which occurrences of context passing can be automatically refactored to the wormhole design pattern. To date, we have not implemented automatic refactoring of detected context passing patterns, we simply present a visual representation of the context passing flows. However, the desire to facilitate refactoring to the wormhole pattern as the next step in the software engineering process led to several design restrictions when deciding how to implement the mining software. ContextFlowMiner detects the following:

- context perception:
  - source contexts : a method that passes a variable that is **not** a parameter as an argument to another method.
  - target contexts : a method that uses a parameter in an expression that is **not** a call to another method.
- context acquisition
  - parameter flow : $m_i \rightarrow n_j$, object that is passed into a method ***m*** as the i^{th} parameter is subsequently passed as the ^{jth} parameter to method *n* that is called within the method body of ***m***. Method m does not use i^{th} parameter in any statements except as an argument to another method call. (this will allow refactoring to the wormhole, where the parameter is removed entirely)
- context-dependent computation
  - abstract portion of target method body containing context-dependent computation

The *ContextFlowMiner* is implemented to mine an entire Java project for context passing as implemented by parameter passing. A limit may be set to denote the minimum length path of a flow in order to avoid trivially short parameter passing sequences. The Java Development Tools (JDT) provided by Eclipse contain perspectives involved in code merging and refactoring. In addition the JDT Core provides a model that allows navigation of the Java source code as an Abstract Syntax Tree. The JDT Core provides an *IMethod* class which represents a method declared in a class or interface. The IMethod will allow access to information about each method, such as its name, its qualified name, its parameters names and types, its return type and any exceptions it may throw. We present a brief list of the steps taken during mining:

1. Create abstract syntax tree (AST) per class.
2. Create *Visitor* to traverse AST, visit each node to look for parameter passing.
3. Detect parameter flow: Current method passes one of its parameters as an argument to another method. The parameter is not used in other statements in the method body, except as an argument to a method call (strictly context *passing*).
4. Create object to represent a parameter flow between methods.
5. Compute chains of parameter flow sequences, factoring in interface implementations and subclass overriding for each method invocation in the chain.
6. Compute callers of source of parameter flow sequences
7. Aggregate groups of parameter flow paths.

We aggregate groups of parameter passing paths that overlap to form a graph. Thus, if there is fan-in or fan-out (i.e. a context is passed into or out of a method parameter along more than one call path), the programmer may view as a group all paths that provide context at some target joinpoint. When refactoring to the wormhole pattern, it is necessary to remove the explicit context parameter from a method parameter list. It is also necessary to refactor the target method body of all references to the context object. Such references must now exist instead in the wormhole advice. Note that we can not remove a parameter from a method unless all callers of the method are refactored. Additionally, we can not remove a parameter along the context passing path if it is used in any statement within a method body (for example, aliasing) other than parameter passing (see step 3 above). Thus, we specifically are mining for parameter passing flows that will allow automated refactoring to the wormhole pattern.

Once the context passing information is mined, we display the information visually to the programmer in a new Eclipse view. For example, Figure 3 shows two overlapping parameter passing flows detected when the plugin was used on the JHotDraw project [11], which has been proposed as a testbed for aspect mining [13]. Figure 3 shows the flow of an object through the sequence: DrawApplication.createDefaultTool → DrawApplication.setDefaultTool$_0$ → DrawApplication.createToolButton$_2$ → ToolButton.ToolButton$_3$.

The subscript on each method indicates the index (0 based) of the method parameter. The view also shows another caller of the ToolButton constructor, DrawApplet.createToolButton. Note that if the programmer wished to refactor only the parameter passing along the first path DrawApplication.createDefaultTool→…→ ToolButton.ToolButton using the WormHole pattern, the path DrawApplet.createToolButton→ToolButton.ToolButton would no longer compile since the wormhole refactoring would have removed the explicit parameter from the ToolButton constructor. Thus, one benefit of the *ContextFlowMiner* tool is to show the programmer all of the incoming callers of a method involved in a parameter passing flow, allowing an informed decision to be made prior to refactoring.

4. CONCLUSION

Numerous types of cross-cutting concerns involve the modularization of a subset of joinpoints that are available along an execution path. There have been techniques proposed that suggest the use of program slicing [2] as well as fan-in [6][8]. In this paper, we focus on a variation of slicing and fan-in that involves data flow through simple parameter passing along the call path. The current implementation of ContextFlowMiner is restricted to context passing that could be replaced with the

wormhole design pattern, thus it is based on evolving Java code to use the cflow pointcut expression. We are investigating how to extend the tool to mine for context passing that can be refactored by the contextual and dflow pointcut expressions.

5. REFERENCES

[1] Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn. *Using AspectC to Improve the Modularity of Path-Specific Customization in Operating System Code*. In Proceedings of the Joint European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-9), pages 88-98. ACM Press, 2001.

[2] T. Cottenier and T. Elrad. *Contextual Pointcut Expressions for Dynamic Service Customization*. In Dynamic Aspects Workshop (DAW), pages 95—99, ACM Press, Mar 2005.

[3] T. Ishio, S. Kusumoto, K. Inoue. *Debugging Support for Aspect-Oriented Program Based on Program Slicing and Call Graph*, In Proceedings of 20th International Conference on Software Maintenance (ICSM2004), pp.178-187, 2004.

[4] R. Laddad. *AspectJ In Action*, Manning Press, 2003.

[5] K. Lieberherr, *Adaptive Object-Oriented Software: The Demeter Method With Propagation Patterns*. PWS, 1996.

[6] M. Marin, A. van Deursen, and L. Moonen. *Identifying Aspects Using Fan-in Analysis*. In Proceedings of the 11th Working Conference on Reverse Engineering (WCRE), IEEE Computer Society, 2004.

[7] H. Masuhara and K. Kawauchi. *Dataflow pointcut in aspect-oriented programming*. In Proceedings of The First Asian Symposium on Programming Languages and Systems (APLAS'03), volume 2895 of Lecture Notes in Computer Science, pages 105--121, Beijing, China, Nov. 2003

[8] D. Shepherd, J. Palm, L. Pollack. *Fast Prototyping and Evaluation of Aspect Mining Analyses via Timna.* Workshop on Aspect Reverse Engineering at the Working Conference on Reverse Engineering 2004.

[9] R. Walker and G. Murphy. *Implicit context: Easing software evolution and reuse*. In Proceedings of the Eighth International Symposium on the Foundations of Software Engineering (FSE-8), 2000.

[10] L. Wall, T. Christiansen and R. Schwatz. *Programming Perl*. O'Reilly and Associates, 2nd edition, 1996.

[11] http://www.jhotdraw.org/

[12] E. Tanter, K.Gybels, M. Denker and A. Bergel, *Context-Aware Aspects*, In Proceedings of the 5th International Symposium on Software Composition (SC 2006), Vienna, Austria, March, LNCS, 2006.

[13] A. van Deursen, M. Marin, L. Moonen. *A Systematic Aspect-Oriented Refactoring and Testing Strategy, and its Application to JHotDraw.* Proceedings of the 2005 Workshop on Modeling and Analysis of Concerns in Software.

[14] D. Hansen and T. Proebsting. *Dynamic Variables.* Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation, Snowbird, Utah, June 2001, 264-273

[15] C. Herzeel, K. Gybels, P. Costanza. *A Temporal Logic Language for Context Awareness in Pointcuts*. In Revival of Dynamic Languages Workshop, ECOOP, 2006.

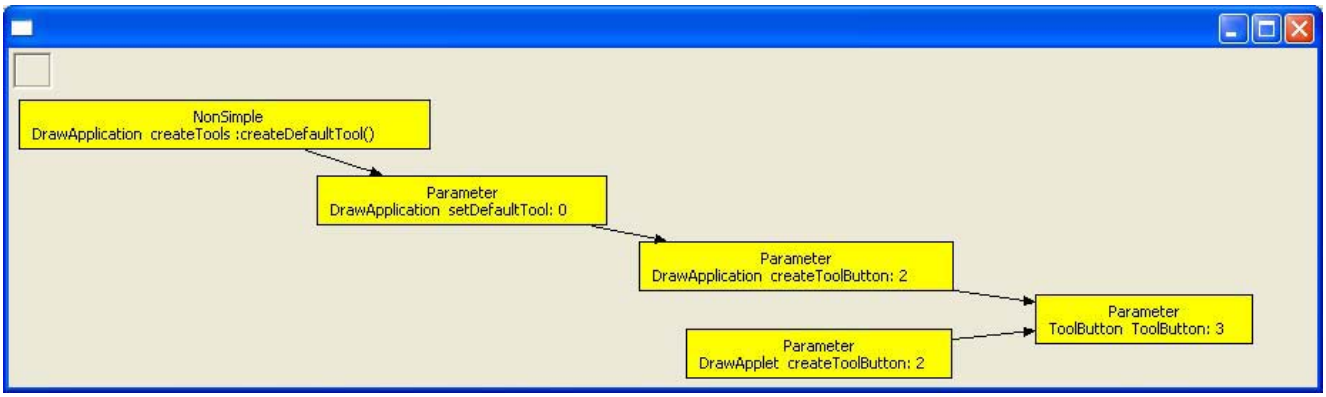

FIGURE 3: JHotDraw parameter passing flow

Quality Measures for Evaluating the Results of Clustering Based Aspect Mining Techniques

Grigoreta Sofia Moldovan
Department of Computer Science
Babeş-Bolyai University
1, Mihail Kogălniceanu Street
Cluj-Napoca, Romania
grigo@cs.ubbcluj.ro

Gabriela Şerban
Department of Computer Science
Babeş-Bolyai University
1, Mihail Kogălniceanu Street
Cluj-Napoca, Romania
gabis@cs.ubbcluj.ro

ABSTRACT

The aim of this paper is to propose a set of new quality measures for evaluating the results of clustering based aspect mining techniques. We have focused on identifying the features that influence the time required by the manual analysis of the results. For aspect mining techniques that use clustering to identify crosscutting concerns, we propose a set of new quality measures for these features. Then, we use these measures to evaluate the results of a clustering based aspect mining technique in two case studies.

1. INTRODUCTION

The identification of crosscutting concerns in legacy systems, or *aspect mining*, is a relatively new research area. Many aspect mining techniques have been proposed so far ([1], [2], [6], [7], [8], [9], [10]). Some of them are dynamic ([1], [2], [10]), some of them are static ([6], [7], [9]), some of them use *formal concept analysis* ([10]), some of them use *fan-in analysis* ([6], [7]), some of them use *clustering* ([2], [7], [9]) and some use *clone detection techniques* ([8]).

We need to identify some measures in order to compare these techniques. So far, the only thing that these techniques have in common, is that the last step of the technique is the manual analysis of the obtained results.

The new quality measures proposed in this paper reflect the time needed for the manual analysis of the results, which should be minimized. In our opinion, from the user's point of view, an aspect mining technique is adequate if:

- R1 it discovers all (or almost all) the crosscutting concerns that exist in the analyzed system;
- R2 all the component parts of a crosscutting concern are grouped together;
- R3 in such a group there are no parts from other (crosscutting) concerns;
- R4 the user has to manually analyze as little as possible of the system source code to discover all crosscutting concerns and the component parts of each crosscutting concern.

The first requirement (R1) is self-explanatory. If the results of the technique are later used for refactoring into aspects, then the user would like to know all the parts that need to be modified (R2). If R2 is satisfied and this group also contains parts from other concerns, and then some refactorings are applied, the behaviour of the new system might change. The third requirement restricts the number of parts that might be modified. The last requirement(R4) is to reduce the time the user needs to analyze the system source code and to ease the integration of the technique with automated AO refactoring tools.

In the clustering-based aspect mining techniques proposed so far ([2], [7], [9]) a software system is considered as a set of methods that are grouped in classes (clusters) using clustering techniques ([3]). A part of these clusters are then analyzed to discover crosscutting concerns.

So far, there were not reported in the literature, quality measures for evaluating the results of clustering based aspect mining techniques. The main contribution of this paper is the definition of a set of new quality measures, in order to evaluate the time required to manually analyze the results obtained by clustering based aspect mining techniques.

The paper is structured as follows. In Section 2 we present the context on which the quality measures are defined. The new quality measures and a small example on how to compute them are presented in Section 3. Section 4 reports some experimental results and Section 5 presents some conclusions and further work.

2. THEORETICAL MODEL

Let $M = \{m_1, m_2, ..., m_n\}$ be the software system, where $m_i, 1 \leq i \leq n$ is a method of the system. We denote by n ($|M|$) the number of methods in the system.

As all existing clustering based aspect mining techniques, we consider in this model the method as the smallest quantity of a crosscutting concern.

Consequently, we consider a crosscutting concern as a set

of methods $C = \{c_1, c_2, ..., c_{cn}\}$, methods that implement this concern. The number of methods in the crosscutting concern C is $cn = |C|$. Let $CCC = \{C_1, C_2, ..., C_q\}$ be the set of all crosscutting concerns that exist in the system M. The number of crosscutting concerns in the system M is $q = |CCC|$. We mention that in our approach we consider the set CCC known a-priori, meaning that we are interested only in evaluating the results, not in the way the crosscutting concerns were discovered. Let $NCCC = M - (\bigcup_{i=1}^{q} C_i)$ be the set of methods from the system M, methods that do not implement any crosscutting concern.

Definition 1. (**Partition of a system M.**)
The set $\mathcal{K} = \{K_1, K_2, ..., K_p\}$ is called a **partition** of the system M iff $1 \leq p \leq n$, $K_i \subseteq M, K_i \neq \emptyset, \forall i \in \{1, 2, ..., p\}$, $M = \bigcup_{i=1}^{p} K_i$ and $K_j \cap K_i = \emptyset$, $\forall i, j \in \{1, 2, ..., p\}, i \neq j$.

In the following we will refer to K_i as the i-th *cluster* of $\mathcal{K}$ and to $\mathcal{K}$ as a *set of clusters*.

Definition 2. (**Optimal partition of a system M.**)
Being given a partition $\mathcal{K} = \{K_1, K_2, ..., K_p\}$ of the system M, $\mathcal{K}$ is called an **optimal partition** of the system M with respect to the set $CCC = \{C_1, C_2, ..., C_q\}$ of all crosscutting concerns, iff:
(1) $p > q$
(2) there exists a permutation $\{\sigma(1), \sigma(2), ..., \sigma(q)\}$ of the set {1,2, ..., q} such that $K_i = C_{\sigma(i)}, \forall i \in \{1, 2, ..., q\}$.

Intuitively, $\mathcal{K}$ is an optimal partition of the system M if all the methods implementing a crosscuting concern C_i ($\forall\ 1 \leq i \leq q$) are in the same cluster K_j ($j \in \{1, 2, ..., p\}$) and they are the only methods in K_j.

Remark 1. If condition (2) in Definition 2 is replaced with a less restrictive one:
(2) there exists a permutation $\{\sigma(1), \sigma(2), ..., \sigma(q)\}$ of the set {1,2, ..., q} such that $C_{\sigma(i)} \subseteq K_i, \forall i \in \{1, 2, ..., q\}$,
then we will call such a partition a **"good" partition**, meaning that for each crosscutting concern, all the methods implementing it are in the same cluster.

3. QUALITY MEASURES

In the following, we propose a set of new quality measures for evaluating the results of clustering based aspect mining techniques.

The first four measures ($DISP$, DIV, PAM and $PREC$) are original quality measures proposed in order to evaluate the results from the aspect mining point of view. The last measure (SSE) is a well-known measure for evaluating the results from the clustering point of view [3]. We intend to find a correlation between the results from the aspect mining and clustering points of view.

We will use the notations described in Section 2. As the result of any clustering technique is a partition $\mathcal{K}$ of the system M and as the result of most aspect mining techniques is a set of methods used to implement crosscutting concerns from the set CCC, all the quality measures will be defined with respect to $\mathcal{K}$ and CCC.

In the following definitions, we will denote by $|A|$ the number of elements (cardinality) of the set A.

3.1 Definitions

Definition 3. (**Dispersion of crosscutting concerns - DISP**)
The dispersion of the set CCC in the partition $\mathcal{K}$, denoted by $DISP(CCC, \mathcal{K})$, is defined as

$$DISP(CCC, \mathcal{K}) = \frac{1}{|CCC|} \sum_{i=1}^{|CCC|} disp(C_i, \mathcal{K}).$$

$disp(C, \mathcal{K})$ is the dispersion of a crosscutting concern C and is defined as:

$$disp(C, \mathcal{K}) = \frac{1}{|D_C|},$$

where $D_C = \{k | k \in \mathcal{K} \text{ and } k \cap C \neq \emptyset\}$. D_C is the set of clusters that contain methods which are also in C.

In our view, $DISP(CCC, \mathcal{K})$ defines the dispersion degree of crosscutting concerns in clusters. For a crosscutting concern C, $disp(C, \mathcal{K})$ indicates the number of clusters that contain methods belonging to C.

Based on Definition 3, $DISP(CCC, \mathcal{K}) \in (0, 1]$. If $disp(C, \mathcal{K}) = 1$, for each $C \in CCC$, meaning that all the methods of C are in the same cluster, then $DISP(CCC, \mathcal{K}) = 1$, otherwise $DISP(CCC, \mathcal{K}) < 1$.

Larger values for $DISP$ indicate better partitions with respect to CCC, meaning that $DISP$ has to be maximized.

Definition 4. (**Diversity of a partition - DIV**)
The diversity of a partition $\mathcal{K}$ with respect to the set CCC, denoted by $DIV(CCC, \mathcal{K})$, is defined as

$$DIV(CCC, \mathcal{K}) = \frac{1}{|\mathcal{K}|} \sum_{i=1}^{|\mathcal{K}|} div(CCC, K_i).$$

$div(CCC, k)$ is the diversity of a cluster $k \in \mathcal{K}$ and is defined as:

$$div(CCC, k) = \frac{1}{|V_k| + \tau(k)}$$

where $V_k = \{C | C \in CCC \text{ and } k \cap C \neq \emptyset\}$ is the set of crosscutting concerns that have methods in k,
and $\tau(k) = \begin{cases} 1 & \text{if } k \cap NCCC \neq \emptyset \\ 0 & \text{if } k \cap NCCC = \emptyset \end{cases}$.

$\tau(k)$ is 1 if the cluster k contains methods that do not implement any crosscutting concern, and 0 otherwise.

$DIV(CCC, \mathcal{K})$ defines the degree to which each cluster contains methods from different crosscutting concerns or methods from other concerns.

Based on Definition 4, $DIV(CCC, \mathcal{K}) \in (0, 1]$. If for each $k \in \mathcal{K}$, $div(CCC, k)$ is 1, meaning that each cluster contains methods from a single crosscutting concern and no

methods from $NCCC$, or only methods from $NCCC$, then $DIV(CCC, \mathcal{K}) = 1$, otherwise $DIV(CCC, \mathcal{K}) < 1$.

Larger values for DIV indicate better partitions with respect to CCC, meaning that DIV has to be maximized.

Definition 5. (**Percentage of analyzed methods for a partition - PAM**)
Let us consider that the partition $\mathcal{K}$ is analyzed in the following order: $K_1, K_2, ..., K_p$.

The percentage of analyzed methods for a partition K with respect to the set CCC, denoted by $PAM(CCC, \mathcal{K})$, is defined as:

$$PAM(CCC, \mathcal{K}) = \frac{1}{|CCC|} \sum_{i=1}^{|CCC|} pam(C_i, \mathcal{K}).$$

$pam(C, \mathcal{K})$ is the minimum percentage of the methods that need to be analyzed in the partition $\mathcal{K}$ in order to discover the crosscutting concern C, and is defined as:

$$pam(C, \mathcal{K}) = \frac{1}{|M|} \sum_{j=1}^{i} |K_j|$$

where $i = min\{t \,|\, 1 \leq t \leq p \text{ and } K_t \cap C \neq \emptyset\}$ is the index of the first cluster in the partition $\mathcal{K}$ that contains methods from C.

$PAM(CCC, \mathcal{K})$ defines the percentage of the minimum number of methods that need to be analyzed in the partition in order to discover all crosscutting concern that are in the system M. We consider that a crosscutting concern was discovered the first time a method that implements it was analyzed.

Based on Definition 5, $PAM(CCC, \mathcal{K}) \in (0, 1]$. If each $C \in CCC$ has one method in the first analyzed cluster K_1 of the partition $\mathcal{K}$, then $PAM(CCC, \mathcal{K}) = \frac{|K_1|}{|M|}$, otherwise $PAM(CCC, \mathcal{K}) > \frac{|K_1|}{|M|}$.

Smaller values for PAM indicate short time for analysis, meaning that PAM has to be minimized.

Definition 6. (**Precision of a clustering based aspect mining technique - PREC**)
Let $\mathcal{T}$ be a clustering based aspect mining technique.

The precision of $\mathcal{T}$ with respect to a partition $\mathcal{K}$ and the set CCC, denoted by $PREC(CCC, \mathcal{K}, \mathcal{T})$, is defined as:

$$PREC(CCC, \mathcal{K}, \mathcal{T}) = \frac{1}{|CCC|} \sum_{i=1}^{|CCC|} prec(C_i, \mathcal{K}, \mathcal{T}).$$

$prec(C, \mathcal{K}, \mathcal{T}) = \begin{cases} 1 & \text{if C was discovered by } \mathcal{T} \\ 0 & \text{otherwise} \end{cases}$ is the precision of $\mathcal{T}$ with respect to the crosscutting concern C.

$PREC(CCC, \mathcal{K}, \mathcal{T})$ defines the percentage of crosscutting concerns that are discovered by $\mathcal{T}$. In all clustering based aspect mining techniques, only a part of the clusters are analyzed, meaning that some crosscutting concerns may be missed.

Based on Definition 6, $PREC(CCC, \mathcal{K}, \mathcal{T}) \in [0, 1]$. If all crosscutting concerns are discovered by $\mathcal{T}$, then $PREC(CCC, \mathcal{K}, \mathcal{T}) = 1$, otherwise $PREC(CCC, \mathcal{K}, \mathcal{T}) < 1$.

Larger values for $PREC$ indicate better partitions with respect to CCC, meaning that $PREC$ has to be maximized.

Definition 7. (**Squared sum error of a partition - *SSE***)
The squared sum error of a partition $\mathcal{K}$, denoted by $SSE(\mathcal{K})$, is defined as:

$$SSE(\mathcal{K}) = \sum_{j=1}^{p} \sum_{i=1}^{n_j} d^2(m_i^j, f_j)$$

where the cluster K_j is a set of methods $\{m_1^j, m_2^j, ..., m_{n_j}^j\}$ and f_j is the centroid (mean) of K_j.

From the point of view of a clustering technique, smaller values for SSE indicate better partitions, meaning that SSE has to be minimized.

Note. We have to make the following remarks:

- It can be proved that $\mathcal{K}$ is an optimal partition with respect to CCC iff $DISP(CCC, \mathcal{K}) = 1$ and $DIV(CCC, \mathcal{K}) = 1$.
- An **ideal** partition is an optimal partition with $PREC$ equal to 1 and a minimum value for PAM.
- An **"almost ideal"** partition is a "good" partition (Remark 1), with $PREC$ equal to 1 and a minimum value for PAM.
- In order to decide if a clustering technique is efficient in aspect mining, we would expect that a good partition from the clustering point of view (considering the SSE measure) would also be adequate from the aspect mining point of view (considering the $DISP$, DIV, PAM and $PREC$ measures).

3.2 Example

In the following, a small example showing how to compute the measures $DISP$, DIV and PAM, is presented. $PREC$ and SSE are not discussed because they depend on the clustering based aspect mining technique or on the clustering technique used.

Let $M = \{m_1, m_2, ..., m_{15}\}$ be a software system with 15 methods, and let $C_1 = \{m_2, m_3\}$ and $C_2 = \{m_1, m_4, m_8\}$ be the crosscutting concerns that exist in the system M ($CCC = \{C_1, C_2\}$).

Let $\mathcal{K} = \{K_1, K_2, K_3, K_4, K_5\}$ be a partition of the system M, where:

$K_1 = \{m_2\}$
$K_2 = \{m_3, m_7, m_8, m_9\}$
$K_3 = \{m_1, m_4, m_5, m_{12}\}$
$K_4 = \{m_6, m_{10}, m_{13}\}$
$K_5 = \{m_{11}, m_{14}, m_{15}\}$

DISP

Using Definition 3, we have to compute $disp(C, \mathcal{K})$ for each $C \in CCC$. The obtained values are shown below.

Crosscutting concern	The set D	*disp*
C_1	$\{K_1, K_2\}$	0.5
C_2	$\{K_2, K_3\}$	0.5

Based on the definition, $DISP(CCC, \mathcal{K}) = 0.5$.

DIV

Using Definition 4, we have to compute $div(CCC, k)$ for each $k \in \mathcal{K}$. The obtained values are shown below.

Cluster	The set V	τ	*div*
K_1	$\{C_1\}$	0	1
K_2	$\{C_1, C_2\}$	1	0.33
K_3	$\{C_2\}$	1	0.5
K_4	$\emptyset$	1	1
K_5	$\emptyset$	1	1

Based on the definition, $DIV(CCC, \mathcal{K}) = 0.76$.

PAM

Using Definition 5, we have to compute $pam(C, \mathcal{K})$ for each $C \in CCC$. For that we have to determine the index of the first analyzed cluster that contains method(s) from C. The obtained values are shown below.

Crosscutting concern	# first cluster	*pam*
C_1	1	0.06
C_2	2	0.33

Based on the definition, $PAM(CCC, \mathcal{K}) = 0.19$.

4. EXPERIMENTAL RESULTS

In [7] we have proposed a clustering based aspect mining approach that uses *k-means* and *hierarchical agglomerative* clustering techniques [3]. In order to group the methods in clusters, we have used the vector space model approach and we have defined two models($\mathcal{M}_1$, $\mathcal{M}_2$). We also reported the obtained results in two case studies: Carla Laffra's implementation of Dijsktra's algorithm [5] and JHotDraw version 5.2 [4].

The results obtained using these techniques were evaluated based on the measures proposed in Section 3.

For lack of space we present in Table 1 the values of the quality measures of the results obtained when only *k-means* was used.

Case study	Model	DISP	DIV	PAM	PREC	SSE
Laffra	$\mathcal{M}_1$	0.75	0.79	0.06	1	3.39
Laffra	$\mathcal{M}_2$	0.75	0.89	0.04	1	4.27
JHotDraw	$\mathcal{M}_1$	0.43	0.82	0.07	0.87	11.96
JHotDraw	$\mathcal{M}_2$	0.42	0.95	0.06	0.87	55.26

Table 1: The values of the quality measures for the two case studies.

After analyzing the obtained results, we have noticed the following:

- In almost all cases there is a correlation between good partitions from the clustering point of view and from the aspect mining point of view.
- The values of $DISP$, DIV and $PREC$ indicate that the vector space models used in our approach should be improved.

5. CONCLUSIONS AND FURTHER WORK

In this paper we have proposed a set of new quality measures for evaluating the results of clustering based aspect mining techniques. We have computed these measures in order to determine the quality of the results obtained by applying the techniques we have proposed in [7]. By analyzing the results, we have identified the possible drawbacks of our clustering approach. The results also indicate that clustering techniques, properly applied, are useful in aspect mining.

Further work can be done in the following directions:

- To evaluate other clustering based aspect mining techniques using the proposed measures.
- To generalize these measures for other aspect mining techniques (like [1], [6], [10]).
- To identify other possible measures for evaluating clustering approaches in aspect mining.
- To improve our approach ([7]) using the quality measures described in this paper.

6. REFERENCES

[1] S. Breu and J. Krinke. Aspect Mining Using Event Traces. In *Proc. International Conference on Automated Software Engineering (ASE)*, pages 310–315, 2004.

[2] L. He and H. Bai. Aspect Mining using Clustering and Association Rule Method. *International Journal of Computer Science and Network Security*, 6(2A):247–251, February 2006.

[3] A. Jain, M. N. Murty, and P. Flynn. Data clustering: A review. *ACM Computing Surveys*, 31(3):264–323, 1999.

[4] JHotDraw Project. http://sourceforge.net/projects/jhotdraw.

[5] C. Laffra. Dijkstra's Shortest Path Algorithm. http://carbon.cudenver.edu/ hgreenbe/courses/dijkstra/ DijkstraApplet.html.

[6] M. Marin, A. van, Deursen, and L. Moonen. Identifying Aspects Using Fan-in Analysis. In *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE2004)*, pages 132–141. IEEE Computer Society, 2004.

[7] G. S. Moldovan and G. Serban. Aspect Mining using a Vector-Space Model Based Clustering Approach. In *Proceedings of Linking Aspect Technology and Evolution Workshop(LATE 2006)*, March 2006.

[8] O. A. M. Morales. Aspect Mining Using Clone Detection. Master's thesis, Delft University of Technology, The Netherlands, August 2004.

[9] D. Shepherd and L. Pollock. Interfaces, Aspects, and Views. In *Proceedings of Linking Aspect Technology and Evolution Workshop(LATE 2005)*, March 2005.

[10] P. Tonella and M. Ceccato. Aspect Mining through the Formal Concept Analysis of Execution Traces. In *Proceedings of the IEEE Eleventh Working Conference on Reverse Engineering (WCRE 2004)*, pages 112–121, November 2004.

Aspect Mining for Aspect Refactoring: An Experience Report

Maximilian Störzer, Uli Eibauer and Stefan Schoeffmann
Universität Passau, Passau, Germany
{stoerzer, eibauer}@fmi.uni-passau.de, stefan.schoeffmann@sdm.de

ABSTRACT

Aspect-Oriented programming currently suffers from one increasingly important problem–while there is an abundance of aspect-oriented languages and systems, only few example programs are publicly available. To lighten this situation, we set out to refactor crosscutting concerns into aspects for Open Source Java systems.

Aspect Mining (AM) is an important enabler of Aspect-Oriented Refactoring (AOR), and this paper reports about our preliminary experience with automatic and manual aspect refactoring. From this experience we formulate interesting research questions for further research.

1. MOTIVATION

Aspect-Oriented Programming has been proposed to address limitations in current programming paradigms called the *tyranny of the dominant decomposition* in literature [8]. While there is an abundance of available languages, currently only few non-trivial examples programs are publicly available.

To address this lack of code we started two projects related to aspect-oriented refactoring in the spirit of related work on (A)JHotDraw [9, 1]. Our projects were originally independent of each other: first, we designed an automatic refactoring tool[7] for Eclipse based on fully automatic aspect mining[1] using DynAMiT[3] and conducted three case studies with this tool. Second we are currently working on a project targeted to refactor the open source Java application HSQLDB (http://hsqldb.org/) using a semantics-guided approach. Both our tool and the refactoring project use AspectJ as target language.

The purpose of our refactoring tool was to easily generate AspectJ programs out of Open Source Java applications. However, this goal turned out to be very ambitious. Nevertheless we learned some important lessons for usability of aspect mining results for automatic refactoring which we report in this paper.

HSQLDB is a medium-size open source project (65 kLoC), implementing an relational database system. HSQLDB comes with a JUnit test suite which we use to guarantee functional equivalence of our system. Clearly the first step for an aspect-oriented refactoring is to find relevant crosscutting concerns which actually can be refactored as aspects. We used manual semantics-guided code inspection supported by FEAT[6] to find relevant crosscutting code.

As we originally did not intend to use these two projects to evaluate aspect mining techniques, we did not perform our case studies using the same projects. However, we argue that the basic observations and results we report here are inherent to the underlying techniques—automatic versus manual aspect refactoring—and are thus an interesting contribution, even if we agree that a more thorough case study on one subject to verify these results is needed.

The contributions of this paper are twofold. First we report our experience from two aspect-refactoring projects–one conducted with automatic, one with manual aspect mining–and derive interesting research questions for aspect mining tools from a comparison of this experience. Second, as for HSQLDB both the Java and the AspectJ version will be available once our project is finished, our effort will also result in an interesting evaluation test case for (new) aspect mining tools. Comparing results of AM tools to the aspects we found by manually analyzing the system might be an interesting benchmark.

2. AUTOMATIC AOR WITH DYNAMIT

DynAMiT is an automatic aspect mining tool based on dynamic program analysis. The tool evaluates traced call sequences to discover repeated patterns, which are then–if certain thresholds in repetition are reached–reported as aspect candidates.

DynAMiT discovers candidates for `before` and `after`-advice for `call` and `execution` joinpoints. For example DynAMiT discovers that each time `f()` is called, `g()` is called immediately after and then suggests that the call to `g()` should be embedded in an `after`-advice to the call to `f()` or, symmetrically, the call to `f()` should be embedded in a `before`-advice to the call to `g()`.

Our automatic aspect refactoring tool uses DynAMiT to find aspect candidates, analyzes its results to figure out if a refactoring is feasible, and, if so, allows to automatically refactor aspect candidates. For our analysis we check if candidates identified by DynAMiT can be moved to an aspect (using AspectJ) without changing program semantics. Therefore, context at the joinpoint has to be available for AspectJ, and values (after the joinpoint and attached advice have been executed in the refactored program version) must be equal to the original values. To implement our analysis, we built our system on the Java refactoring framework available in Eclipse, and used human interaction if we could not derive a result.

We conducted three case studies to evaluate our tool, one based on the source code of DynAMiT itself, one by analyzing the Jakarta Commons Codecs project, and finally one by analyzing the ANTLR parser generator framework. For each system, we analyzed if it is possible to semi-automatically refactor aspects from the automatically derived results presented by DynAMiT.

We soon discovered that DynAMiT–as a dynamic analysis tool–has two important disadvantage for automatic refactoring. Repeated call sequences are analyzed without any structural information about the underlying calls. This means that DynAMiT also reports repeated patterns if a call is governed by an `if`-statement or part of a loop, i.e. the found patterns strongly depend on the

[1]DynAMiT analyzes call relations without human interaction to derive candidates for crosscutting concerns. We use the term *automatic aspect mining* for comparable non-interactive techniques.

Case Study	DynAMiT						CommonCodecs						ANTLR					
Algorithm	Crosscutting			Basic			Crosscutting			Basic			Crosscutting			Basic		
Refactorizability	√	?	×	√	?	×	√	?	×	√	?	×	√	?	×	√	?	×
before call	0	0	2	0	0	3	0	0	0	0	0	2	2	0	6	1	0	43
after call	0	0	1	0	0	4	0	0	0	0	0	3	0	0	11	0	1	49
before execution	0	0	2	1	0	4	2	0	0	4	0	2	4	1	5	22	1	44
after execution	0	1	2	1	2	3	0	2	0	1	4	0	3	7	16	12	9	118
Coverage (statement)	56,9 %						96,8 %						25,7 %					

√ Semantics Preserving Refactoring feasible, × Refactoring failed due to Dependences, ? Semantical Change depends on Method Call

Table 1: Some Numbers: Candidates discovered by DynAMiT and their Refactorizability using our Tool

test suite used to generate the traces. If a insufficient test suite is used[2], extraction in an aspect is only possible and meaningful in the traces cases, and will produce different system semantics otherwise. This could be prevented if complicated pointcut expressions using the `if`-designator are generated to create functionally equivalent aspects. Consequently analysis of *control dependences* is important to check if an aspect candidate can be automatically refactored. However, we refrained from extracting such aspect candidates as such complex conditions more likely are an indicator for false positives (no quantified statement).

Second, method calls are not the only statements. We often experienced situations, where several assignments preceded the first call in a method. In these cases, the first call can only be extracted in a `before execution` advice if it is guaranteed that the joinpoint context is not modified by the above assignments, i.e. program semantics have to be equivalent if the call is moved to the method entry (code motion problem). That means *data-flow constraints* can considerably restrict refactorizability of crosscutting code.

Third, necessary joinpoint context sometimes is simply not available as no respective joinpoint exists in the target language (e.g. local variables or literals used in a call we want to extract in an aspect are not available for AspectJ). This means that *language limitations* also hinder aspect-oriented refactoring.

Please note that we did not examine the results of DynAMiT for *semantical soundness*, but only examined if they allow automatic refactoring resulting in a semantically equivalent program. For our case studies, most results had to be discarded. Table 1 gives some details on the case studies we performed. Consider for example the first column group labeled DynAMiT. Here, we got 8 aspect candidates in total when using the more strict "Crosscutting" algorithm (some of them symmetric). From these, only 1 candidate could be semi-automatically refactored. Our system has no pointer-analysis to safely approximate the effects of method calls, and thus does not allow us to automatically decide about refactorizability in all cases. We thus ask the user in such cases, using the *refactoring view* known from the Eclipse Java refactorings. These cases are counted in the '?' column. For the other 7 cases our tool found direct control and data flow dependences or was not able to access the context, all of which prevented refactoring. Note that DynAMiT is based on dynamic analysis, and thus the test coverage of the suite the analysis is based on is a very important issue in this context. The last line thus reports the coverage of the test suites we used for our case studies.

The Commons Codes system produced similar results. For ANTLR we got 55 advice candidates, and could only refactor 9 of these results. However, in 8 other cases a refactoring might have been possible, although our limited analysis could not decide this automatically. To summarize the above observation, control and data flow properties as well as language limitations are very important to decide if automatic refactoring of a crosscutting concern is possible. Recent work [2] of the author of DynAMiT also recognized these problems and added additional static analysis support.

To connect this observation with aspect-mining tools used for refactoring, analyzing and reporting data and control dependences can be used to (i) reduce the false positive rate and (ii) give additional information useful for programmers when they actually refactor code. Hence, analyzing refactorizability of candidates could be an additional criterion for the quality of an aspect-mining tool and might serve to give additional feedback to the user.

A second observation is that automatic syntax-based aspect mining tends to produce many false positives. Such tools *identify crosscutting code*–but crosscutting code not necessarily is due to a *crosscutting concern*. Crosscutting code is an indication for a crosscutting concern, but not a sufficient criterion; the decisive criterion is the *actual semantics* of the crosscutting code. Additionally the user still has to figure out all those identified patterns that actually belong to the same concern manually and thus should be encapsulated in the same aspect. So there is also a *mismatch in granularity* for purely syntax-based automatic techniques.

DynAMiT reported several candidates where we could not identify an semantical concern inducing the crosscutting code. From our perspective it is very hard if not impossible to distinguish between "accidentally" crosscutting code and crosscutting code due to an actual crosscutting concern without additional semantical information. This seems to be a general restriction of automatic syntax-based mining approaches.

3. MANUAL AM USING FEAT

Compared to the above study with automatic aspect-mining based on DynAMiT, we used a semantics-driven manual approach for HSQLDB. To find aspects here we based our analysis on the list of 'standard aspects' introduced in Laddad's book "AspectJ in Action" [5] and then used FEAT to discover relevant code locations. When becoming familiar with the source code we also found some *application specific aspects*, for example trigger firing or checking constraints before certain operations are performed.

To support manual system analysis, FEAT proved to be very effective. FEAT is a user guided cross referencing tool and allows to quickly discover code locations referencing some method or field. What we basically did–slightly simplified–was to discover potentially interesting classes–like e.g. `Tracing` or `Cache`–and then to use FEAT to discover where these classes are referenced. These references then have to be eliminated and replaced with an aspect to conduct the aspect-oriented refactoring.

We have finished the aspect mining phase and begun to actually implement the aspect-oriented refactoring. Our observations

[2]Note that for semantics-preservation, even a statement coverage of 100 % is *not* sufficient.

reported here are thus based on the aspects we identified, but we can not yet report if the aspect-oriented refactoring will actually be successful[3] in all cases. Refactoring in this case is manual, not automatic. Thus we are not as restricted in the ways we can refactor a system as in the above tool project.

For our analysis we manually discovered the starting point for a search, but this manual analysis was guided by our aspect catalog. We then used FEAT to find the locations where a crosscutting concern is tangled with other modules. Thus instead of using syntactical or low level properties of the system, we used a semantical approach. We started with a certain concern we expected to find in the system in mind, and tried to retrieve the code locations for its implementation. Compared to the above automatic study per definition no semantically questionable aspect candidates can occur. While this reduces the false positive rate, a considerable amount of *manual code inspection and analysis* (although supported by FEAT) was necessary to fulfill this task.

However, even for these manually identified crosscutting concerns, refactoring in general is not straightforward. Some of the problems we encountered are similar to the problems we discovered in the DynAMiT case study. FEAT also discovers calls to be extracted within a loop, or governed by an `if`-statement. Although this crosscutting code results from an *actual crosscutting concern*, refactoring these calls is nevertheless problematic. One strategy we use in this case is to pre-process the code (i.e. extract some code in methods if this is adequate) to allow a subsequent aspect-oriented refactoring. From a software engineering point of view, this cannot be a general solution as it easily jeopardizes system structure.

Our semantic catalog-guided approach was successful in discovering many standard aspects in the HSQLDB code base, including Tracing, Caching and Authentication/Authorization. To summarize, we think that augmenting aspect-mining tools with semantical information might be a fruitful approach for aspect mining. For example one might identify a set of classes related to a semantical concern–like e.g. a `Logger` class–and then demand that all reported candidates have to be related to one of these classes.

We will also illustrate these observations—both for aspect mining and refactoring—with an example. For HSQLDB, we identified the *tracing concern* as a crosscutting concern and its implementing crosscutting code. Tracing has been considered a standard crosscutting concern since the invention of AOP, so the aspect mining for this specific concern was relatively easy and straightforward following the strategy described above. Refactoring this concern and extracting its code into an aspect however was far from trivial. The problem is that *custom tracing* in an existing system cannot be formulated with a quantified statement like "On each method entry, log the method name and the parameter values.". The *implementation is* rather considerably more *customized for each method* to capture the values of interest within this method—including local variables and their changes e.g. within loops. Such customized tracing policies are very hard to capture in an aspect. We did this as an example for some classes, and to succeed we had to: create a *common trace format* (i.e. the system now produces a different output!), refactor *loop bodies to helper methods* (arranged pattern problem!), or *"promote" local variables to fields* (locality?). To make a long story short: the resulting implementation is—from a software engineering point-of-view—at least *questionable*.

However there are also positive examples. We identified *pooling*, also a standard crosscutting concern according to Laddad, as an aspect that can be refactored easily without the problems mentioned above. The source of HSQLDB contains a class `ValuePool` which contains relevant pooling logic. When an `Integer`, a `Long`, `String`, `Double`, `Date`, or `BigDecimal`-object is needed, the corresponding access method in the pool is explicitly invoked. Calls to these accessors occured at approximately 250 locations in the source code. As a result of these scattered calls we observed a high coupling between the classes containing these calls and class `ValuePool`.

For refactoring, these explicit calls to the value pool were replaced by the corresponding constructor calls (e.g. `new Integer()`). We then advised the constructors with `around`-advice which invokes the appropriate pool methods without calling `proceed`. This approach has several advantages compared to the purely object-oriented variant: As the pooling aspect is implemented as a separate aspect, the coupling due to the explicit pool invocations has disappeared (only the pooling aspect knows about the relation between class `ValuePool` and the remaining system). Second, the aspect now can be removed from the core program without any additional base changes. Finally, the aspect captures additional 190 code locations that failed to invoke the value pool before, as we used wildcards for the respective constructors to specify the pointcuts. To summarize, the aspect-oriented implementation in this case is clearly superior compared to the original version.

Although not all *refactorings* were successful, our semantic catalog-guided *mining* approach was nevertheless very successful in *discovering* many standard aspects in the HSQLDB code base, including Tracing, Caching, Pooling and Authentication/Authorization. To summarize, we think that augmenting aspect-mining tools with semantical information might be a fruitful approach for aspect mining.

4. LESSONS LEARNED

Most automatic aspect-mining approaches we are aware of are either based on finding repeated patterns in call sequences/traces/etc. or on finding duplicated code.

From our experience, actually refactoring advice candidates found by such tools has to deal with several important problems.

Control Dependences: Control dependences can easily lead to false positives, for example if a method call is always triggered in an available test suite used for analysis, but not necessarily triggered every time. While in some cases candidate code governed by an `if`-statement can be refactored to advice using an equivalent `if` pointcut designator, this is not true in general. Loops are an even more important problem.

Data Flow Restrictions: Advice cannot be attached to arbitrary code positions. If code should be moved to an aspect, it is possible that this code has to be moved e.g. to the beginning or the end of a method. This is of course not possible in general.

Arranged Pattern Problem: The code to be refactored in general uses some values from its context. These values thus always have to be accessible for the aspect language in order to allow a refactoring. Especially for AspectJ this is often problematic.

It is tempting to argue that any refactoring is possible, if we only use enough purely object-oriented refactorings to remove problematic control and data-flow dependences and make necessary joinpoint context available to our aspect language. However, this will result in another problem called the *arranged pattern problem* in [4]. Code is transformed only to allow advice application, but *not* to create well-defined, easy to understand, reusable, and evolvable methods. As a consequence, software quality degrades. This might be a language problem rather than an aspect mining issue, however suggesting such refactorings is problematic nonetheless.

Semantics vs. Syntactical Properties: The most important question: *Did we really find a crosscutting concern?* Even if

[3]I.e. if it is possible to refactor an identified concern or if the concern is too tightly coupled thus preventing refactoring.

syntactically a tool can derive an aspect candidate, is this candidate also semantically a valid concern? The use of *utility classes* is a good example. In general functionality of such classes is called from several parts of the system, however the modularity of the system is fine and nobody would argue that references to `java.lang.Math` show a crosscutting concern.

When looking at aspect mining results it is tempting to dismiss non-refactorizable candidates as false positives, although this is not true in general. However, if a crosscutting concern has an implementation too tightly coupled with the system, refactoring may not be a valid option anyway.

So is a purely semantic approach as we used for HSQLDB the method of choice? This method clearly has the advantage that we do not have to deal with many false positives. However, refactoring the code to advice faces the same problems as the automatic syntax-based aspect mining tools before. This justifies that simply removing non-refactorizable candidates from a result set is not a valid option, i.e. *'refactorizability' is no criterion to rule out candidates*—but it might help to order them by relevance for refactoring.

Semantic-based aspect mining also has another important disadvantage: We started our analysis based on the standard aspects catalog provided by Laddad. By only following this technique we will per definition only find aspects we know about–but *never new ones*. This is clearly a strength of syntax-based mining tools.

As a challenge to the aspect-mining community it might be interesting to create aspect mining tools which help programmers to identify standard crosscutting concerns in a given system. Of course such a tool could not be automatic, but compared to FEAT more automation might considerably help programmers trying to refactor standard crosscutting concerns. The main improvement we suggest is to also use semantical information to guide automatic tools when retrieving aspect candidates. Not all repeated method call sequences are advice candidates, but maybe those referencing a certain class are. Not each piece of duplicated code is a unrefactored advice, but maybe code referencing certain fields. This approach would combine automatic support from automatic aspect mining with the semantic guidance useful to avoid false positives.

While such a tool is interesting for a practitioner in the field trying to refactor an existing application based on a catalog of known aspects, it might also be interesting to develop tools designed to identify new aspects. Theses tools however are designed for researchers, who set out to better understand the nature of aspects in general, and also to extend the aspect catalog.

For evaluation of aspect-mining, both suggested tool categories have a considerably different profile and need different evaluation strategies. Tools targeted to discover standard aspects need appropriate systems where a refactored aspect-oriented and an original version exist. Based on these two versions, quality of the results is accessible. Evaluating tools designed to discover new aspects is considerably harder. The above strategy is not useful in this case.

5. CONCLUSION

In this paper we discussed the results of a fully automatic aspect mining tool in contrast to a manual aspect mining approach.

From our experience many aspect candidates proposed by the automatic aspect mining tool are not useful for an automatic refactoring, as language restriction, i.e. un-accessible context, control dependences, i.e. calls to-be-extracted which are embedded in loops/governed by `if`-statements, or data-dependences, i.e. modifications of parameter values prior to calls to-be-extracted, prevent refactoring. Reviewing these problems for particular cases often also raises doubt if the corresponding code is actually part of the implementation of a crosscutting concern.

Our second study based on manual aspect mining was successful to discover standard aspects, but failed to reveal any new/application specific aspects. This is a general weakness of this approach. While here per definition no false positives occur (either a valid concern can be found or not), refactoring the found crosscutting code might not be recommendable due to a high coupling with the base system.

To improve result quality for aspect mining tools, we suggest to build two kinds of tools: (i) aspect mining tools guided by a catalog of well-known crosscutting concerns to assist software engineers in actually refactoring existing systems and (ii) less restricted automatic mining tools designed to help researchers find completely new aspects. For the first category of tools refactorizability might be a good criterion to prioritize mining results.

Using projects like AJHotDraw and HSQLDB as case studies (once our project is finished) seems to be a good way to evaluate category (i) aspect-mining tools. We encourage researchers to use their tools to also refactor other projects as case studies and make the resulting aspect-oriented systems publicly available.

Acknowledgments

Thanks to the anonymous reviewers and Daniel Wasserrab for their valuable and interesting comments on this paper.

6. REFERENCES

[1] Dave Binkley, Mariano Ceccato, Mark Harman, Filippo Ricca, and Paolo Tonella. Automated Refactoring of Object Oriented Code into aspects. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 27–36, Washington, DC, USA, 2005. IEEE Computer Society.

[2] Silvia Breu. Extending Dynamic Aspect Mining with Static Information. In *5th International Workshop on Source Code Analysis and Manipulation (SCAM 2005), Budapest, Hungary*, October 2005.

[3] Silvia Breu and Jens Krinke. Aspect Mining Using Event Traces. In *19th International Conference on Automated Software Engineering (ASE 2004)*, pages 310–315, September 2004.

[4] Kris Gybels and Johan Brichau. Arranging language features for more robust pattern-based crosscuts. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 60–69, New York, NY, USA, 2003. ACM Press.

[5] Ramnivas Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Co., Greenwich, CT, USA, 2003.

[6] Martin P. Robillard and Gail C. Murphy. Concern graphs: finding and describing concerns using structural program dependencies. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 406–416, New York, NY, USA, 2002. ACM Press.

[7] Stefan Schöffmann. Semi-automatisches Aspect Refactoring–Tool-Entwicklung und Fallstudie auf Basis bestehender Aspect Mining Tools. Master's thesis, Universität Passau, Innstraße 32, 94032 Passau, Germany, Dezember 2004.

[8] Peri Tarr, Harold Ossher, William Harrison, and Jr. Stanley M. Sutton. N degrees of separation: multi-dimensional separation of concerns. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 107–119, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[9] Arie van Deursen, Marius Martin, and Leon Moonen. AJHotDraw: A showcase for refactoring to aspects. In *In Proceedings AOSD Workshop on Linking Aspect Technology and Evolution*, 2005.

TUD-SERG-2006-012
ISSN 1872-5392