A Survey of Automated Code-Level Aspect Mining Techniques

Andy Kellens[1]*, Kim Mens[2], and Paolo Tonella[3]

[1] Programming Technology Lab, Vrije Universiteit Brussel
Pleinlaan 2, B-1050 Brussels, Belgium
akellens@vub.ac.be

[2] Département d'Ingénierie Informatique
Université catholique de Louvain
Place Sainte Barbe 2, B-1348 Louvain-la-Neuve, Belgium
kim.mens@uclouvain.be

[3] ITC-irst, Centro per la Ricerca Scientifica e Tecnologica
Via Sommarive 18, 38050 Trento, Italy
tonella@itc.it

Abstract. This paper offers a first, in-breadth survey and comparison of current aspect mining tools and techniques. It focuses mainly on automated techniques that mine a program's static or dynamic structure for candidate aspects. We present an initial comparative framework for distinguishing aspect mining techniques, and assess known techniques against this framework. The results of this assessment may serve as a roadmap to potential users of aspect mining techniques, to help them in selecting an appropriate technique. It also helps aspect mining researchers to identify remaining open research questions, possible avenues for future research, and interesting combinations of existing techniques.

1 Introduction

Aspect-oriented software development (AOSD) tries to solve the problem of separating the core functionality of a software system from concerns that have a more system-wide behaviour and that cut across the primary decomposition of the software system. This problem is sometimes referred to as the "tyranny of the dominant decomposition" [1]. To overcome this prevalent decomposition [2], the AOSD paradigm provides new language constructs, like advices and pointcuts [3], which allow cross-cutting concerns to be written down in a new kind of module named *aspect*.
Almost ten years after its initial conception, this technology has left the research lab and is starting to be adopted by industry, which poses new interesting research problems. Just like the industrial adoption of the object-oriented paradigm in the early nineties led to a need for migrating legacy software systems to an object-oriented solution — triggering

* Ph.D. scholarship funded by the "Institute for the Promotion of Innovation through Science and Technology in Flanders" (IWT Vlaanderen).

a boost of research on software reverse engineering, reengineering, restructuring and refactoring — the same is currently happening to the aspect-oriented paradigm.
The reasons for wanting to migrate a legacy system to an aspect-oriented solution are multiple. Due to the presence of crosscutting concerns, legacy systems tend to contain many symptoms of duplicated code, *scattering* of concerns throughout the entire system and *tangling* of concern-specific code with that of other concerns. Using aspect-oriented technology, these cross-cutting concerns can be cleanly separated from the base code, which becomes oblivious of them. This is supposed to make the system easier to understand, maintain and evolve.
However, manually applying aspect-oriented techniques to a legacy system is a difficult and error-prone process. Due to the large size of such systems, the complexity of the implementation, the lack of documentation and knowledge about the system, there is a need for tools and techniques that can help software engineers in locating or documenting the cross-cutting concerns in legacy systems or for more automated tools to discover such concerns, as well as for tools and methodologies to refactor the discovered cross-cutting concerns into aspects.
The study and development of such approaches is the objective of the emerging research domains of aspect mining and aspect refactoring. Whereas *aspect mining* is the activity of discovering cross-cutting concerns that potentially could be turned into aspects, *refactoring to aspects* is the activity of actually transforming these potential aspects into real aspects in the software system. (See Figure 1.)

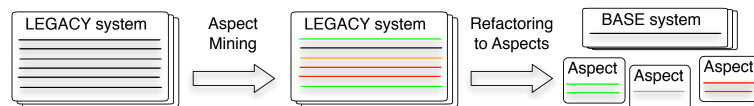

Fig. 1. Migrating a legacy system to an aspect-oriented system

This paper focusses on the activity of aspect mining and conducts a survey of existing code-level techniques, tools and methodologies that have been designed to aid a software engineer in discovering aspect candidates in a legacy system. A multitude of such techniques have recently been proposed, making it hard for potential users to decide which technique is most appropriate for their needs. This survey may serve as a roadmap to them by providing a taxonomy and comparison of currently existing aspect mining techniques, as well as some of their limitations and underlying assumptions.
The survey is also expected to be useful to the aspect mining research community. Although still in its infancy, this research area has recently seen a proliferation of approaches and techniques, inspired by several different research domains. Future research efforts will necessarily be devoted to comparing and combining the alternative approaches. This survey can be seen as a first step in that direction. It provides guidance in

determining groups of similar techniques and highlighting their different underlying preconditions and properties.

To the authors' knowledge, this paper is the first published survey of existing aspect mining techniques. Its main contributions to the state of the art are:

- the definition of a set of criteria of comparison;
- the derivation of a taxonomy for the classification of techniques;
- the discussion of the properties of the existing techniques, according to the classification framework;
- the identification of future research directions in the area.

The paper is organized as follows: Section 2 defines aspect mining and positions it with respect to other aspect discovery approaches. Section 3 gives an overview of existing aspect mining techniques. The classification criteria are introduced in Section 4 and applied to the surveyed techniques in Section 5. The outcome of the classification is discussed in Section 6. Before concluding the paper in Section 8, a description of related research areas is given in Section 7.

2 Aspect Discovery

As mentioned above, the process of migrating a legacy system into a system using aspects consists of two steps: the discovery of aspect candidates and the refactoring of (some of) these candidates into aspects. In this survey we investigate techniques and tools that aid a developer in discovering possible aspects. This is not a trivial task, due to the size and complexity of current-day software systems and the lack of explicit documentation on the cross-cutting concerns present in those systems.

Three major kinds of aspect discovery approaches can be distinguished :

Early aspect discovery techniques. Traditionally, AOSD has focussed mostly on the software life-cycle's implementation phase. Research on 'early aspects' tries to discover aspects in earlier phases of the software life-cycle [4], such as requirements and domain analysis [5–7] or architecture design [8]. Identifying and managing early aspects not only helps to improve modularity in requirements and architectural design, but many early aspects eventually find their way into the code as implementation aspects.

In the context of legacy systems, where requirements and architecture documents are often outdated, obsolete or no longer available, early aspect discovery techniques cannot be applied and approaches that focus on source code are thus potentially more promising.

Dedicated browsers. A second class of approaches are the advanced special-purpose code browsers that aid a developer in manually navigating the source code of a system to explore cross-cutting concerns. Although the primary goal of these approaches is not to explicitly mine for aspects, but rather to document and localise cross-cutting concerns in order to maintain and evolve a system, these dedicated browsers can be used to identify aspects in a system as well.

Typically, a user of such a browsing approach starts out with a 'seed' of a concern, a starting point in the code, and uses the browser to

further explore this concern. To do so the browser may propose other hotspots in the code which are related to the concern or provide the user with a query language to manually traverse the concern. Examples of such approaches are Concern Graphs [9], Intensional Views [10], Aspect Browser [11], (Extended) Aspect Mining Tool [2, 12] and Prism [13].

Aspect mining techniques *automate the process of aspect discovery* and propose their user one or more aspect candidates. To this end, they reason about the source code of the system or about data that is acquired by executing or manipulating the code. All techniques seem to have at least in common that they search for symptoms of cross-cutting concerns, using either techniques from data mining and data analysis like formal concept analysis and cluster analysis, or more classic code analysis techniques like program slicing, software metrics and heuristics, clone detection and pattern matching techniques, dynamic analysis, and so on.

In this survey we focus only on this third category of automated code-level tools and techniques that assist a developer in the activity of mining for cross-cutting concerns in an existing system. We define *aspect mining* as follows:

> *Aspect mining* is the activity of discovering those cross-cutting concerns that potentially could be turned into aspects, from the source code and/or run-time behaviour of a software system. We refer to such concerns as 'aspect candidates'.

3 Overview of Aspect Mining Techniques

This section offers a detailed overview of the different automated code-level aspect mining approaches that have been proposed over the last few years.

3.1 Analysing recurring patterns of execution traces

Breu and Krinke propose an aspect mining technique named *DynAMiT* (Dynamic Aspect Mining Tool) [14], which analyses program traces reflecting the run-time behaviour of a system in search of recurring execution patterns. To do so, they introduce the notion of *execution relations* between method invocations. Consider the following example of an event trace, where the capitals represent method names:

```
B() {
      C() {
            G() {}
            H() {}
          }
    }
A() {}
```

Breu and Krinke distinguish between four different execution relations: outside-before (e.g., B is called before A), outside-after (e.g. A is called after B), inside-first (e.g., G is the first call in C) and inside-last (e.g., H is the last call in C). Using these execution relations, their mining algorithm discovers aspect candidates based on recurring patterns of method invocations. If an execution relation occurs more than once, and recurs uniformly (for instance, every invocation of method B is followed by an invocation of method A), it is considered to be an aspect candidate. To ensure that the aspect candidates are sufficiently cross-cutting, there is an extra requirement that the recurring relations should appear in different 'calling contexts'. Although this approach is inherently dynamic, the authors have repeated the experiment using control-flow-graphs [15] to calculate the call relations statically. Breu also reports on a hybrid approach [16] where the dynamic information is complemented with static type information in order to remove ambiguities and improve on the results of the technique.

3.2 Formal concept analysis

Formal concept analysis (FCA) [17] is a branch of lattice theory which, given a set of objects and attributes describing those objects, creates *concepts*, i.e., maximal groups of objects that have common attributes. These concepts are organised into a lattice, according to the partial order associated with attribute (or equivalently object) set inclusion.

Formal concept analysis of execution traces Tonella and Ceccato [18] developed *Dynamo*, an aspect mining tool which applies FCA to execution traces in order to discover possible aspects. When analysing a system using Dynamo, an instrumented version of the system is executed on a number of use cases, manually derived from the software documentation and/or from a high level description of the main functionalities. The output of this execution is a number of execution traces. These traces are then analysed using FCA: the use cases are the objects of the FCA algorithm, while the methods which get invoked during the execution of a use case are the attributes. In the resulting lattice, all concepts are selected which contain traces from exactly one use-case. These are regarded as aspect candidates if the following (automatically verified) conditions hold:

- Scattering: The specific attributes (methods) of the concept belong to more than one class.
- Tangling: Different methods from the same class are specific to more than one use-case specific concept.

Formal concept analysis of identifiers Tourwé and Mens [19] propose an alternative aspect mining technique which relies on FCA. Unlike the Dynamo tool discussed above, Tourwé and Mens's *DelfSTof* tool analyses the source code of a system (experiments have been conducted on Smalltalk code [19] and on Java code [20]). Their approach

performs an identifier analysis using the FCA algorithm. The assumption behind this approach is that interesting concerns in the source code are reflected by the use of naming conventions in the classes and methods of the system. As input to the FCA algorithm, the classes and methods in the system are used as objects. As attributes, the FCA algorithm uses substrings generated from the classes and methods' names. For instance, a class named `QuotedCodeConstant` is split into the strings 'Quoted', 'Code' and 'Constant'. Substrings with little meaning, like 'a', 'with', ... are discarded from the results. The resulting concepts consist out of maximal groups of classes and methods which share a maximal number of substrings. After having filtered out many unimportant concepts automatically, a significant number of concepts remain which need to be inspected manually. Apart from being able to detect a number of programming idioms, design patterns and certain refactoring opportunities [21], the same approach can be used for aspect mining purposes [19] by restricting the concepts to those that are crosscutting (i.e. the involved methods and classes belong to at least two different class hierarchies).

3.3 Natural language processing on source code

Similar to the previous approach, Shepherd et al. [22] propose a technique that is based on the assumption that cross-cutting concerns are often implemented by the rigorous use of naming and coding conventions. Their approach uses natural language processing (NLP) information as an indicator for possible aspect candidates. They report on an experiment in which they use an NLP technique called *lexical chaining* [23] in order to find groups of related source-code entities which represent a cross-cutting concern. Lexical chaining will output, given a collection of words as input, chains of words which are semantically strongly related. In order to create the chains, the algorithm requires a semantical distance measure between each combination of words. To this end, Shepherd et al. used the WordNet [24] database, in combination with information about the parts of speech of each word, to calculate the semantical path between two words. In order to mine for cross-cutting concerns, they apply the chaining algorithm to the comments, method names, field names and class names of the system they are analysing. A user of their approach needs to manually inspect the resulting chains in order to select likely aspect candidates.

3.4 Detecting unique methods

Gybels and Kellens [25, 26] propose the use of heuristics to mine for cross-cutting concerns. They observe that, in pre-AOP days, cross-cutting concerns were often implemented in an idiomatic way. Certain of these idioms can be regarded as "symptoms" of aspect candidates. An example of such an idiom is the implementation of a cross-cutting concern by means of a single entity in the system which is called from numerous places in the code (for instance, a 'logging' entity which is called from throughout the code). To detect instances of this pattern, Gybels and Kellens propose the "Unique Methods" heuristic which is defined as:

“A method without a return value which implements a message implemented by no other method.”

After calculating all unique methods in a system, sorting them according to the number of times a method is called, and filtering out irrelevant methods (like for instance *accessor* and *mutator* methods), the user has to manually inspect the resulting methods in order to find suitable aspect candidates. Regardless of the simplicity of this approach, the authors demonstrated the applicability of their technique by detecting typical aspects like tracing, update notification and memory management in the context of a Smalltalk image.

3.5 Clustering of related methods

Hierarchical clustering of similar method names Shepherd and Pollock [27] report on an experiment in which they used agglomerative hierarchical clustering [28] to group related methods. This technique starts by putting each method in a separate cluster and then recursively merges clusters for which the distance between the methods is smaller than a certain threshold. They implemented this technique as part of an aspect-oriented IDE named *AMAV* (Aspect Miner and Viewer), which allows for easy adaptation of the distance measure used by the algorithm. For an initial experiment they used a simple distance measure opposite proportional to the common substring length of the names of the methods. This mining algorithm is used in combination with the viewing tool of the IDE which not only lists all the clusters which were found, but also consists out of a *cross-cutting pane* which displays the methods related to a cluster as well as an *editor pane*, in which the class context of a particular method is displayed.

Clustering based on method invocations He and Bai [29] propose another aspect mining technique based on cluster analysis. They start from the assumption that if the same methods are called frequently from within different modules, this may be a good indication that a hidden cross-cutting concern is present. As input for the clustering algorithm, a set of methods is given along with a distance measure based on the Static Direct Invocation Relationship (SDIR) between the methods. This distance measure varies between 0 and 1 and represents the dissimilarity of the methods. Methods which are closely related (i.e. which get called frequently together) will have distance approximating 0, while the distance between methods which are never or seldom called together will be close to 1.

3.6 Fan-in analysis

Marin et al. [30] noticed that many of the well-known cross-cutting concerns exhibit a high fan-in. They propose using a fan-in metric in order to discover cross-cutting concerns in source code. They define the fan-in of a method m as the number of distinct method bodies which can invoke m. Because of polymorphism, a call to a method m contributes to

the fan-in of all methods refining m, as well as method m itself. Their mining algorithm comprises out of the following steps:

- Calculating the fan-in metric for all methods in the system.
- Filtering the results: next to filtering *accessor* and *mutator* methods, as well as utility methods like for instance `toString()`, the number of considered methods is also limited by only considering the methods with a fan-in value higher than a certain threshold.
- Manually analysing the remaining methods.

The authors present an experiment in which cross-cutting concerns were mined with a high precision: one third of all methods with high fan-in were seeds leading to an aspect. Moreover, 60% of the remaining two thirds were removed automatically.

3.7 Clone detection

The symptom of 'code duplication' may be a good indicator of cross-cutting concerns in the source code of a system: because the cross-cutting concerns could not be cleanly modularised, certain parts of the implementation show high levels of duplicated code. Two techniques rely on this observation to mine for aspect candidates.

Detecting aspects using PDG-based clone detection A first technique, presented by Shepherd et al. [31] and implemented as a tool they call *Ophir*, makes use of *program dependence graphs* (PDG) to detect possible aspects. In a PDG, each statement in the code is represented by a node; the edges of the graph consist of control or data dependence relations between the statements. By comparing PDGs [32, 33], this technique is able to recognise code duplication in the beginning of a method (i.e. aspect candidates for a 'before' advice). After filtering and coalescing the resulting PDGs, a number of possible aspect candidates remain.

Using AST- and token-based clone detection Bruntink et al. also make use of clone detection techniques to mine for aspects. In [34, 35], they compare *token-based* [36] clone detection, which is based on a lexical analysis of the source code, with *AST-based* [37] clone detection, which takes the parse tree of the source code into account. Both techniques output a number of *clone classes*, i.e. groups of code fragments which are considered to be clones of each other. They applied the clone detection techniques to a large C program in which the different cross-cutting concerns were annotated by a developer. In order to measure the effectiveness of the techniques, Bruntink et al. empirically compare the resulting clone classes with the manual documentation of the cross-cutting concerns. Bruntink reports on a refinement of this work [38], in which a number of metrics for the clone classes are described which can be used to filter the results of the clone detection techniques.

4 Criteria of Comparison

We now present a set of criteria that will allow us, in Section 5, to compare the aspect mining techniques listed in Section 3. We compiled this set by focussing on the variabilities of the different techniques. As such we intended to obtain a taxonomy that supports potential users of aspect mining techniques to select an adequate technique, and that helps aspect mining researchers to understand the differences between their own and existing techniques. The taxonomy contrasts the different restrictions each technique imposes on the input data, the kinds of analysis which are used, the degree of automation and the scalability of each technique. These criteria form an initial comparative framework that may still evolve, following new developments in the field of aspect mining.

Static versus dynamic data *What kind of data does the technique analyse?* Does it analyse input data which can be obtained by statically analysing the code, or dynamic information which is obtained by executing the program, or both?

Token-based versus structural/behavioural analysis *Which kind of analysis does the technique perform?* We distinguish between:

- **Token-based** Lightweight lexical analysis of the program: sequences of characters, regular expressions, etc.
- **Structural/Behavioural** Structural and behavioural analysis of the program: parse trees, type information, message sends, etc.

Granularity *What is the level of granularity of the mined aspect candidates?* While some techniques discover aspects at the level of methods, others work more fine-grained by considering individual statements or code fragments as part of the aspect candidates.

Tangling and Scattering *What symptoms of aspects does the aspect mining technique look for?* Does it explicitly look for symptoms of scattering, tangling, or both? Cross-cutting concerns are characterised by high tangling and scattering.

User involvement *What kind of user involvement is required in order to mine for aspects?* What effort does the technique require from its user? Does the user have to manually browse through all results of the technique in order to indicate viable aspect candidates? Is there additional input required from the user during the mining process?

Largest system *On what size of system has the technique been applied?* Even though a technique might behave well and exhibit interesting properties when applied to small examples, this does not imply that the same holds when the technique is tried on larger software systems. Problems may arise from the computational complexity, the need for user involvement, degradation of accuracy with size, etc. Thus, validation on large software systems may be used as an indicator of scalability.

Empirical Validation *To what degree have existing techniques been validated quantitatively on real-life cases?* For the validations done, has it been reported how many of the known aspects were found and how many of those reported were false positives?

Preconditions *What (explicit or implicit) conditions must be satisfied by the concerns in the program under investigation in order for a particular mining technique to find suitable aspect candidates?*

5 Assessment

Based on the criteria introduced in Section 4, we compare the different aspect mining techniques summarised in Section 3. To gain space, we abbreviate the names of the techniques used , as shown in Table 1.

Abbreviated name	Short description of the technique	Section
Execution patterns	Analysing recurring patterns of execution traces	3.1
Dynamic analysis	Formal concept analysis of execution traces	3.2
Identifier analysis	Formal concept analysis of identifiers	3.2
Language clues	Natural language processing on source code	3.3
Unique methods	Detecting unique methods	3.4
Method clustering	Hierarchical clustering of similar method names	3.5
Call clustering	Clustering based on method invocations	3.5
Fan-in analysis	Fan-in analysis	3.6
Clone Detection (PDG-based)	Detecting aspects using PDG-based clone detection	3.7
Clone Detection (AST/token-based)	Detecting aspects using AST-based and token-based clone detection	3.7

Table 1. List of techniques that were compared

For each of the studied techniques, Table 2 shows the kind of data (static or dynamic) analysed by that technique, as well as the kind of analysis performed (token-based or structural/behavioural).

	Kind of input data		**Kind of analysis**	
	static	dynamic	token-based	structural/behavioural
Execution patterns	X	X	-	X
Dynamic analysis	-	X	-	X
Identifier analysis	X	-	X	-
Language clues	X	-	X	-
Unique methods	X	-	-	X
Method clustering	X	-	X	-
Call clustering	X	-	-	X
Fan-in analysis	X	-	-	X
Clone detection (PDG)	X	-	-	X
Clone detection (token)	X	-	X	-
Clone detection (AST)	X	-	-	X

Table 2. Kind of input data and kind of analysis of each technique

Most techniques work on statically available data. 'Dynamic analysis' reasons about execution traces and thus requires executability of the code under analysis. Only 'Execution patterns' works with *both* kinds of input, since both a static version which uses control-flow graphs, and a dynamic version which uses execution traces, exist. As for the kind of reasoning, four techniques perform a token-based analysis of the input data. 'Identifier Analysis' and 'Method Clustering' reason about the names of the methods in a system only. The 'Language Clues' approach is token-based because it reasons about individual words which appear in the program's source code. The four token-based techniques all rely on the assumption that cross-cutting concerns are often implemented by the rigorous use of naming conventions. The seven other techniques reason about the input at a structural or behavioural level.

	Granularity		**Symptoms**	
	method	code fragment	scattering	tangling
Execution patterns	X	-	X	-
Dynamic analysis	X	-	X	X
Identifier analysis	X	-	X	-
Language clues	X	-	X	-
Unique methods	X	-	X	-
Method clustering	X	-	X	-
Call clustering	X	-	X	-
Fan-in analysis	X	-	X	-
Clone detection	-	X	X	-

Table 3. Granularity of and symptoms looked for by each technique

Table 3 summarises the finest level of granularity (methods or code fragments) of the different techniques, and whether they look for symptoms of scattering and/or tangling. With a few exceptions, the typical granularity of the techniques surveyed is at method level. Therefore, most techniques output several sets of methods, each representing a potential aspect seed. Only the three 'Clone detection' techniques detect aspect code at the level of code fragments and can therefore provide more fine-grained feedback on the code that needs to be put into the advice of the refactored aspect. All techniques use scattering as the basic indicator of the presence of a cross-cutting concern. Only 'Dynamic analysis' takes *both* scattering and tangling into account, by requiring that the methods which occur in a single use-case scenario are implemented in multiple classes (scattering), but also that these methods occur in multiple use-cases and thus are tangled with other concerns of the system.

Technique	Largest case	Size case	Empirically validated
Execution patterns	Graffiti	3100 methods/82KLOC	-
Dynamic analysis	JHotDraw	2800 methods/18KLOC	-
Identifier analysis	JHotDraw	2800 methods/18KLOC	-
Language clues	PetStore	10KLOC	-
Unique methods	Smalltalk image	3400 classes/66000 methods	-
Method clustering	JHotDraw	2800 methods/18KLOC	-
Call clustering	Banking example	12 methods	-
Fan-in analysis	JHotDraw TomCat 5.5 API	2800 methods/18KLOC 172KLOC	- -
Clone detection (PDG)	TomCat	38KLOC	-
Clone detection (AST/token)	ASML C-Code	20KLOC	X

Table 4. An assessment of the validation of each technique

To provide more insights into the validation of the techniques, Table 4 mentions the largest case on which each technique has been validated, together with the size of that case, and whether the results have been evaluated quantitatively (for example, how many known aspects were actually reported, how many false positives and negatives were reported, and so on). While the size of the largest analysed system is significant for most of the studied techniques (only 'Call Clustering' was applied to a toy example only), empirical validation of the results was almost always

neglected. It is also worth noting that 4 out of 9 techniques have been validated on the same case: JHotDraw.

One important criterion to help selecting an appropriate technique to mine a given system for aspects is what implicit or explicit assumptions that technique makes about how the crosscutting concerns are implemented. Table 5 summarises these assumptions in terms of preconditions that a system has to satisfy in order to find suitable aspect candidates with a given technique.

Technique	Preconditions on crosscutting concerns in the analysed program
Execution patterns	Order of calls in context of crosscutting concern is always the same.
Dynamic analysis	At least one use case exists that exposes the crosscutting concern and another one that does not.
Identifier analysis	Names of methods implementing the concern are alike.
Language clues	Context of concern contains keywords which are synonyms for the crosscutting concern.
Unique methods	Concern is implemented by exactly one method.
Method clustering	Names of methods implementing the concern are alike.
Call clustering	Concern is implemented by calls to same methods from different modules.
Fan-in analysis	Concern is implemented in separate method which is called a high number of times, or many methods implementing the concern call the same method.
Clone detection	Concern is implemented by reusing a certain code fragment.

Table 5. What conditions does the implementation of the concerns have to satisfy in order for a technique to find viable aspect candidates?

'Identifier Analysis', 'Method Clustering', 'Language Clues' and 'Token-based clone detection' all rely on the assumption that developers rigorously made use of naming conventions when implementing the crosscutting concerns. 'Execution Patterns' and 'Call Clustering' assume that methods which often get called together from within different contexts are candidate aspects. The fan-in technique assumes that crosscutting concerns are implemented by methods which are called many times (large footprint), or by methods calling such methods.

Technique	User Involvement
Execution patterns	Inspection of the resulting "recurring patterns".
Dynamic analysis	Selection of use cases and manual interpretation of results.
Identifier analysis	Browsing of mined aspects using IDE integration.
Language clues	Manual interpretation of resulting lexical chains.
Unique methods	Inspection of the unique methods; eased by sorting on importance.
Method clustering	Browsing of mined aspects using IDE integration.
Call clustering	Manual inspection of resulting clusters.
Fan-in analysis	Selection of candidates from list of methods, sorted on highest fan-in.
Clone detection	Browsing and manual interpretation of the discovered clones.

Table 6. Which kind of user involvement do the different techniques require?

Table 6 summarises the kind of involvement that is required from the user. None of the existing techniques works fully automatic. All techniques require that their users browse through the resulting aspect candidates in order to find suitable aspects. Some require that the users

supply appropriate input, like for instance the 'Dynamic Analysis' technique which expects as input also a number of use-cases.

We combined all these tables into a single taxonomy, depicted in Figure 2, that could serve as an initial roadmap to aspect miners and researchers. Each of the nine considered techniques is represented by a small rectangle. The four larger rectangles distinguish 'static' from 'dynamic' techniques, and 'method-level' techniques from techniques that report 'method fragments' as seeds. The two rounded rectangles on the left partition the considered techniques into 'token-based' techniques and "structural/behavioral' ones.

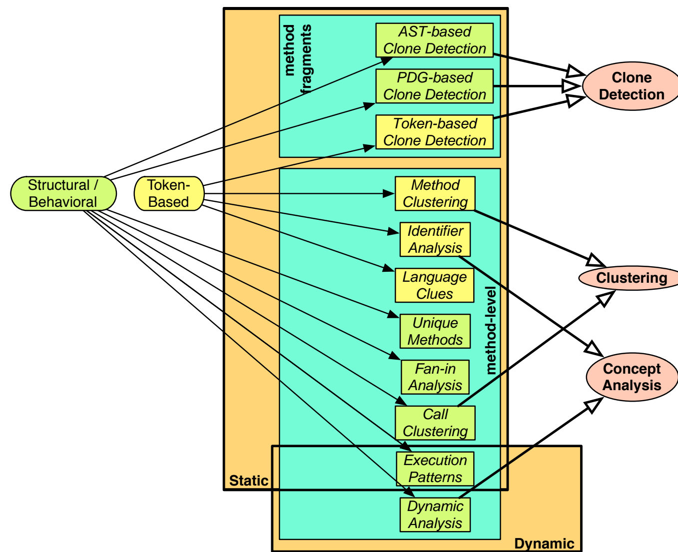

Fig. 2. An initial taxonomy of Automated Code-Level Aspect Mining Techniques

The largest rectangles, separating static from dynamic techniques, represent an extremely relevant criterion for practitioners and researchers, since it entails different requirements on the input (source code vs. executable system) and different interpretations of the output (conservative vs. partial results). This is discussed more thoroughly in the next section.

It is interesting to observe that all known dynamic techniques are structural/behavioral and work at method-level, whereas the static techniques can be divided into 'method-level' and 'method-fragments', as well as based on the kind of information used (i.e., 'token-based' vs. 'structural/behavioural'). This is relevant information for tool developers or practitioners, who might have knowledge about the best aspect indicators to use (lexical oriented or structure/behaviour based) or who may have certain demands about the granularity of the results.

Each technique listed in Figure 2 is uniquely classified by each of the three criteria described above (except 'Execution patterns', which relies on both static and dynamic analysis). The ellipses on the right show some additional taxonomic properties that group some (but not all) of the techniques. More specifically, they specify the basic algorithm underlying the technique: 'Concept analysis', 'Clustering' or 'Clone detection'.

6 Discussion

In this section we discuss some of the lessons we have learned from our comparison of automated code-level aspect mining techniques.

Static versus dynamic data. By relying on static information only, most techniques impose little requirements on the program under analysis. Often, the static information required by these techniques can be computed even for software systems that do not form a complete executable or, in some cases, for code that does not even compile or parse. Dynamic techniques impose heavier constraints by requiring both compilation and execution, as well as an appropriate execution environment. For a user of aspect mining techniques, this can have a significant impact on the choice of the used technique.

The only technique which applies both static and dynamic information is 'Execution patterns' [16]. While for this technique the static information is used to perform a more fine-grained analysis than with its purely dynamic variant, for the other techniques we discussed, a combination of static and dynamic analysis does not immediately seem to provide any obvious advantages. Future aspect mining techniques however may combine static information with dynamic analysis. A common problem with dynamic analysis is that, for large systems, the data provided by tracing the execution of the system might be huge. One possible way to limit this enormous amount of data could be to use static information to restrict the number of places in the code which will be instrumented, thus resulting in smaller traces.

Granularity. Most techniques retrieve aspect candidates as sets of methods that pertain to a crosscutting concern. This kind of granularity works reasonably well if the entire method implements the cross-cutting behaviour, and thus the places in the code where this method gets called can be considered as the joinpoints where the refactored aspect needs to intervene. However, for concerns like e.g. parameter checking, the crosscutting behaviour is not localised into a single method-call, but is instead implemented by a 'pattern' in the code, which is spread throughout multiple statements. In such cases, the user has to provide additional effort in analysing the results of the aspect mining technique in order to highlight the code fragments that are part of the cross-cutting concerns, or use a technique which works at the granularity of method statements.

Tangling and Scattering. Both tangling and scattering have been presented as indicators of cross-cutting concerns. While all techniques take scattering into account, and try to approximate it by for instance requiring calls to cross-cutting behaviour to originate from different class hierarchies, none of the techniques, with the exception of 'Dynamic analysis', look for symptoms of tangling. This can be explained by the fact

that any heuristic for tangling needs high-level information about the different concerns in a system. Since 'Dynamic Analysis' makes use of use-case scenarios, it can take tangling into account by requiring that methods of different classes should be specific to a use-case scenario. As such information seems hard to approximate without external information, it seems unlikely that techniques which analyse source code only can easily take tangling into account. However, when information on artefacts from the earlier phases of the software engineering process are available, these may be exploited to provide a heuristic for tangling.

Empirical validation. In order to perform a quantitative comparison of the studied techniques, empirical validation is of fundamental importance. In most reported studies, however, it was skipped and replaced with a more qualitative result assessment. This is due to the intrinsic difficulty of such a validation: it is hard to define (a priori) the set of relevant aspects to be discovered and to decide (a posteriori) which reported aspects are wrong. Nevertheless, it is impossible for this discipline to make further progress without such an empirical validation. This requires the ability to measure the precision and recall of the results, both in terms of the reported aspects and in terms of the discovered aspect seeds (code entities assigned to the aspect candidates).

In addition to the precision and recall metrics, user studies should be conducted to empirically validate the results. End users of the aspect mining techniques (e.g., the programmers of a system on which the aspect mining is being applied) should be involved in order to evaluate the actual usefulness and usability of each proposed technique. In addition to studies in an industrial context, replication of such studies with students, in classroom settings, would be fruitful too. Such studies might provide important indications of the actual needs that emerge in the execution of a real task of migration toward AOSD. This might in turn steer the research on aspect mining (and refactoring) approaches.

Scalability. Although we have mentioned the largest system on which each technique was validated, it is impossible at this time to make hard claims regarding the scalability of the techniques we studied. While the size of the system can give some indication of whether a technique might scale, and while some techniques were applied to large industrial systems, we cannot make general claims based on the limited data provided by the authors of the different aspect mining techniques. In order to properly assess the scalability of a technique, one would not only have to take into account the time complexity of a technique with respect to the input size, but also the amount of user involvement required for applying that technique. Techniques which require vast amounts of effort to browse through may be less cost-effective. Measuring the amount of required user involvement is strongly related with the need for more empirical validation, as metrics like precision and recall are necessary to quantify this property.

Preconditions. All of the techniques make *different* assumptions about how cross-cutting concerns are implemented in the system, in order for the technique to find viable aspect candidates (see Table 5). Since the assumptions of the techniques we studied seem quite complementary, and each technique thus aims at discovering a different flavour of implemen-

tation of cross-cutting concerns, it is advisable for users of aspect mining techniques to apply multiple techniques to the same system. This way, aspects that are missed by one technique because they do not exhibit a particular symptom, may be detected by another technique. The assessment of the current aspect mining techniques we presented in this paper can serve as a roadmap for a developer to select which techniques might be applicable to mine for aspect candidates in a given system.

Common benchmark. There is a strong need to compare the quality of the different aspect mining techniques that have been proposed in literature. Only very few approaches provide a detailed analysis of their effectiveness. Most techniques are presented only as a proof-of-concept in which it is demonstrated that useful aspects are found. And even for those techniques that have presented more detailed results, they cannot necessarily be compared with others either because they were performed on a different case or because they were presented in a non-compatible format, possibly using different metrics. Therefore, in order to obtain better insights into the strengths and weaknesses of known aspect mining approaches, it is advisable to validate the different techniques on a common case-study and using a common set of well-defined metrics.

JHotDraw [39] seems to be a good candidate for becoming a common benchmark for aspect mining techniques. Ceccato et al. [20] already described an experiment in which they used this case to qualitatively compare three different aspect mining techniques: 'Dynamic analysis', 'Fan-in analysis' and 'Identifier analysis'. In addition, JHotDraw is currently being reworked to an aspect-oriented version, AJHotDraw [40], which is supposed to be behaviourally consistent with JHotDraw itself. The results of mining aspects on JHotDraw could be compared with those aspects actually present in AJHotDraw.

7 Related research areas

From our description of the different techniques in Section 3, it became clear that the field of aspect mining is strongly related to and inspired by a number of other research fields, of which we recall the most important ones in this section. A closer study of these related fields may provide useful ideas to advance the state of the art in aspect mining.

Data Mining. A number of the techniques we discussed make use of data mining algorithms like *cluster analysis* and *formal concept analysis*. This does not come as a surprise, as in the past *data mining techniques* have already been successfully applied on large-scale data sets to retrieve groups of elements which conceptually belong together. This research domain is quite extensive however, and may contain many other approaches which may be ideal candidates for being used as the basis of aspect mining techniques.

Software Comprehension. Aspect mining is closely related to techniques that aid developers in understanding a piece of software. While the goal of *software comprehension techniques* is more general than discovering cross-cutting concerns in legacy code, the results obtained in this field can lead to interesting insights concerning aspect mining.

Program Analysis. In a similar way, knowing that some *program analysis techniques* (for example, *clone detection*) have already been successfully applied to aspect mining, it might be worthwhile considering other program analysis techniques like *slicing* and *metrics* for the purpose of aspect mining.

Re(verse) Engineering. There has been quite some research on how to re(verse)-engineer an ill-structured software system to one that is better structured (for example, with a nice object-oriented design and well-defined architecture). Variants of some of these re(verse) engineering techniques may prove useful for aspect mining as well.

Concept and Feature Location. Existing approaches to locate high level concepts or features (i.e., user-triggered functional requirements) in the source code may be used for the location of cross-cutting functionalities as well, assuming these are known to the programmers. Thus, they have good potential of combination with the aspect mining techniques surveyed in this paper, as was the case for Eisenbarth et al.'s feature location method [41] and Dynamo [18].

To the authors' knowledge, the present work is the first attempt to provide a survey of currently existing aspect mining techniques. Its main contribution to the aspect mining literature is a tentative framework for the classification of the existing techniques into a coherent taxonomy. Research contributions in the area of aspect mining are being produced at an extremely high rate (further works have appeared since the date of submission of this paper), so we cannot be exhaustive. Rather, we aim at defining a framework that can be reused, possibly with adaptations, whenever a new technique needs to be compared with the existing ones.

8 Conclusion

In this paper we presented a survey of existing automated code-level aspect mining techniques. To compare these techniques we proposed a comparative framework and taxonomy which allowed us to discriminate among the different techniques. From this comparison we learned important lessons that can be used by practitioners when selecting an aspect mining technique and that can serve as input to improve the state of the art in this research area.

Aspect mining users are likely to have some knowledge about the system under analysis. They might know whether a set of well-defined use-cases is available and can be employed to isolate relevant concerns or if it is better to rely on the source code alone. This discriminates dynamic versus static analyses. Moreover, they might have some information about the presence of adhered naming conventions throughout the system, which enables token-oriented techniques. Additional knowledge, such as the occurrence of code duplication, might be relevant to decide on the granularity of the technique to use and to select the basic algorithm exploited by the mining technique.

The main contribution of this work to the research area of aspect mining is the definition of a (preliminary) classification framework, that can be used to position each new technique proposed in the area, and to

compare it with the existing ones along relevant dimensions. Other contributions came out of the discussion of the taxonomy. The main future directions that emerged from this study are the need for empirical, comparative evaluations and the opportunity for developing combined techniques. Indeed, since every technique relies on different assumptions and uses different underlying analysis techniques, the studied techniques are highly complementary, which suggests the possibility of several useful combinations. More specifically, it could be worthwhile to:

- combine techniques that rely on static and dynamic analysis;
- combine token-oriented and structural/behavioural techniques;
- extend techniques that work at the granularity of methods with techniques that work at the level of code fragments;
- improve the results of techniques that only look for symptoms of scattering, by taking into account the phenomenon of tangling too;
- combine techniques that rely on different underlying assumptions, thus enabling the discovery of different kinds of aspects.

A natural follow-up to this study would be a more in-depth comparison of the results obtained by the different techniques, based on empirical validation. To enable a quantitative assessment, we need a common set of benchmark programs against which the techniques can be compared, as well as a common set of metrics for measuring the precision and recall of the produced results. Involvement of end-users of the tools to assess the quality of the produced results is also important, considering some of the mined aspect candidates could be refactored eventually into implementation aspects.

Acknowledgments

The authors are grateful to Mariano Ceccato, Marius Marin and Tom Tourwé, to our anonymous reviewers and to other members of the aspect mining community, for the valuable comments they provided on earlier versions of this paper.

References

1. Tarr, P., Ossher, H., Harrison, W., Stanley M. Sutton, J.: N degrees of separation: multi-dimensional separation of concerns. In: International Conference on Software Engineering (ICSE), IEEE Computer Society Press (1999) 107–119
2. Hannemann, J., Kiczales, G.: Overcoming the prevalent decomposition in legacy code. In: Workshop on Advanced Separation of Concerns, International Conference on Software Engineering (ICSE). (2001)
3. Laddad, R.: AspectJ in Action: Practical Aspect-Oriented Programming. Manning Publications (2003)
4. Baniassad, E., Clements, P.C., Araujo, J., Moreira, A., Rashid, A., Tekinerdogan, B.: Discovering early aspects. IEEE Software **23**(1) (2006) 61–70

5. Baniassad, E., Clarke, S.: Theme: An approach for aspect-oriented analysis and design. In: International Conference on Software Engineering (ICSE), Washington, DC, USA, IEEE Computer Society Press (2004) 158–167
6. Rashid, A., Sawyer, P., Moreira, A.M.D., Araújo, J.: Early aspects: A model for aspect-oriented requirements engineerin. In: Joint International Conference on Requirements Engineering (RE), IEEE Computer Society Press (2002) 199–202
7. Tekinerdogan, B., Aksit, M.: Deriving design aspects from canonical models. In Demeyer, S., Bosch, J., eds.: Workshop Reader of the 12th European Conference on Object-Oriented Programming (ECOOP). Lecture Notes in Computer Science, Springer-Verlag (1998) 410–413
8. Bass, L., Klein, M., Northrop, L.: Identifying aspects using architectural reasoning. Position paper presented at Early Aspects 2004: Aspect-Oriented Requirements Engineering and Architecture Design, Workshop of the 3rd International Conference on Aspect-Oriented Software Development (AOSD) (2004)
9. Robillard, M.P., Murphy, G.C.: Concern graphs: Finding and describing concerns using structural program dependencies. In: International Conference on Software Engineering (ICSE 2002), ACM Press (2002) 406–416
10. Mens, K., Poll, B., González, S.: Using intentional source-code views to aid software maintenance. In: International Conference on Software Maintenance (ICSM'03), IEEE Computer Society Press (2003) 169–178
11. Griswold, W., Kato, Y., Yuan, J.: Aspect browser: Tool support for managing dispersed aspects. In: Workshop on Multi-Dimensional Separation of Concerns in Object-oriented Systems. (1999)
12. Zhang, C., Jacobsen, H.: Extended aspect mining tool. http://www.eecg.utoronto.ca/∼czhang/amtex (2002)
13. Zhang, C., Jacobsen, H.A.: Prism is research in aspect mining. In: OOPSLA, ACM (2004)
14. Breu, S., Krinke, J.: Aspect mining using event traces. In: Automated Software Engineering (ASE). (2004)
15. Krinke, J., Breu, S.: Control-flow-graph-based aspect mining. In: 1st Workshop on Aspect Reverse Engineering. (2004)
16. Breu, S.: Towards hybrid aspect mining: Static extensions to dynamic aspect mining. In: 1st Workshop on Aspect Reverse Engineering. (2004)
17. Ganter, B., Wille, R.: Formal Concept Analysis: Mathematical Foundations. Spring-Verlag (1999)
18. Tonella, P., Ceccato, M.: Aspect mining through the formal concept analysis of execution traces. In: Working Conference on Reverse Engineering (WCRE). (2004)
19. Tourwé, T., Mens, K.: Mining aspectual views using formal concept analysis. In: Source Code Analysis and Manipulation Workshop (SCAM). (2004)
20. Ceccato, M., Marin, M., Mens, K., Moonen, L., Tonello, P., Tourwé, T.: A qualitative comparison of three aspect mining techniques. In: International Workshop on Program Comprehension (IWPC 2005), IEEE Computer Society Press (2005) 13–22
21. Mens, K., Tourwé, T.: Delving source-code with formal concept analysis. Elsevier Journal on Computer Languages, Systems & Structures (2005) To appear.
22. Shepherd, D., Tourwé, T., Pollock, L.: Using language clues to discover crosscutting concerns. In: Workshop on the Modeling and Analysis of Concerns. (2005)

23. Morris, J., Hirst, G.: Lexical cohesion computed by thesaural relations as an indicator of the structure of text. Computational Linguistics **17**(1) (1991) 21–48
24. Budanitski, A.: Semantic distance in wordnet: an experimental, application-oriented evaluation of five measures. (2001)
25. Gybels, K., Kellens, A.: An experiment in using inductive logic programming to uncover pointcuts. In: First European Interactive Workshop on Aspects in Software. (2004)
26. Gybels, K., Kellens, A.: Experiences with identifying aspects in Smalltalk using 'unique methods'. In: Workshop on Linking Aspect Technology and Evolution. (2005)
27. Shepherd, D., Pollock, L.: Interfaces, aspects and views. In: Linking Aspect Technology and Evolution (LATE) Workshop. (2005)
28. Karanjkar, S.: Development of graph clustering algorithms. Master's thesis, University of Minnesota (1998)
29. He, L., Bai, H., Zhang, J., Hu, C.: Amuca algorithm for aspect mining. In: Software Engineering and Knowledge Engineering (SEKE). (2005)
30. Marin, M., van Deursen, A., Moonen, L.: Identifying aspects using fan-in analysis. In: Working Conference on Reverse Engineering (WCRE), IEEE Computer Society (2004) 132–141
31. Shepherd, D., Gibson, E., Pollock, L.: Design and evaluation of an automated aspect mining tool. In: International Conference on Software Engineering Research and Practice. (2004)
32. Komondoor, R., Horwitz, S.: Using slicing to identify duplication in source code. In: International Symposium on Static Analysis, Springer-Verlag (2001) 40–56
33. Krinke, J.: Identifying similar code with program dependence graphs. In: Working Conference on Reverse Engineering (WCRE'01), IEEE Computer Society Press (2001) 301–309
34. Bruntink, M., Deursen, A.v., Engelen, R.v., Tourwé, T.: An evaluation of clone detection techniques for identifying crosscutting concerns. In: International Conference on Software Maintenance (ICSM 2004), IEEE Computer Society Press (2004)
35. Bruntink, M., van Deursen, A., van Engelen, R., Tourwè, T.: On the use of clone detection for identifying crosscutting concern code. IEEE Transactions on Software Engineering **31**(10) (2005) 804–818
36. Baker, B.: On finding duplication and near-duplication in large software systems. In: Working Conference on Reverse Engineering (WCRE 1995), IEEE Computer Society Press (1995) 86–95
37. Baxter, I., Yahin, A., Moura, L., Sant' Anna, M., Bier, L.: Clone detection using abstract syntax trees. In: International Conference on Software Maintenance (ICSM 1998), IEEE Computer Society Press (1998)
38. Bruntink, M.: Aspect mining using clone class metrics. In: 1st Workshop on Aspect Reverse Engineering. (2004)
39. Brant, J.: Hotdraw. Master's thesis, University of Illinois (1992)
40. van Deursen, A., Marin, M., Moonen, L.: AJHotDraw: A showcase for refactoring to aspects. In: Workshop on Linking Aspect Technology and Evolution. (2005)
41. Eisenbarth, T., Koschke, R., Simon, D.: Locating features in source code. IEEE Transactions on Software Engineering **29**(3) (2003) 195–209