Applying Semantic Analysis to Feature Execution Traces

Adrian Kuhn, Orla Greevy and Tudor Gîrba
Software Composition Group
University of Bern, Switzerland
{akuhn, greevy, girba}@iam.unibe.ch

Abstract

Recently there has been a revival of interest in feature analysis of software systems. Approaches to feature location have used a wide range of techniques such as dynamic analysis, static analysis, information retrieval and formal concept analysis. In this paper we introduce a novel approach to analyze the execution traces of features using Latent Semantic Indexing (LSI). Our goal is twofold. On the one hand we detect similarities between features based on the content of their traces, and on the other hand we categorize classes based on the frequency of the outgoing invocations involved in the traces. We apply our approach on two case studies and we discuss its benefits and drawbacks.

Keywords: reverse engineering, dynamic analysis, semantic analysis, features, feature-traces, scenarios, static analysis.

1 Introduction

Many reverse engineering approaches to software analysis focus on static source code entities of a system, such as classes and methods [5, 16]. A static perspective considers only the structure and implementation details of a system. Using static analysis alone we are unable to easily determine the roles of software entities play in the features of a system and how these features interact. Without explicit relationships between features and the entities that implement their functionality, it is difficult for software developers to determine if their maintenance changes cause undesirable side effects in other parts of the system.

Several works have shown that exercising the features of a system is a reliable means of correlating features and code [7, 24]. In previous works [9, 10], we described a feature-driven approach based on dynamic analysis, in which we extract execution traces to achieve an explicit mapping between features and software entities like classes and methods. Our definition of a feature is a unit of behavior of a system.

Dynamic analysis implies a vast amount of information, which makes interpretation difficult. We introduce a novel approach that uses an information retrieval technique, namely Latent Semantic Indexing (LSI) [4], to analyze the traces and their relationship to the source code entities. LSI takes as an input a set of *documents* and the *terms* used, and returns a similarity space from which similarities between the documents are ascertained.

In a previous work, we built a reverse engineering approach to cluster the source code entities based on their semantic similarities [13]. In this paper we apply our approach on dynamic information. In other words we use the traces of features as the *text corpus* and we sample this corpus in two different ways to show the generality of our approach.

1. To identify similar features, we use as a document the trace and the method calls involved in the trace as the terms of the document.

2. To identify similarities between classes, we use the classes that participate in feature execution as documents, and all method calls found in the traces outgoing from a class as the terms of the document.

Structure of the paper. We start by introducing the terminology we use to describe and interpret dynamic information. In Section 3 we give an overview of LSI. In Section 4 we describe the details of our approach. In Section 5 we report on the two case studies conducted. We summarize related work in Section 7. Section 8 outlines our conclusions and future work.

2 Feature Terminology

In this section we briefly outline the feature terminology we use. The terms here are based on our previous work [9].

We establish the relationship between the features and software entites by exercising the features or *usage scenarios* and capturing their execution traces, which we refer to as *feature-traces*. A *feature-trace* is a sequence of

runtime events (*e.g.*, object creation/deletion, method invocation) that describes the dynamic behavior of a feature.

We define the measurements $NOFRC$ to compute the # feature-traces that reference a class and FC to compute a characterization of a class in terms of how many features reference it and how many features are currently modeled.

- *Not Covered (NC)* is a class that does not participate to any of the features-traces of our current feature model.

$$(NOFC = 0) \rightarrow FC = 0$$

- *Single-Feature (SF)* is a class that participates in only one feature-trace.

$$(NOFC = 1) \rightarrow FC = 1$$

- *Group-Feature (GF)* is a class that participates in less than half of the features of a feature model. In other words, group-feature classes/methods provide functionality to a group of features, but not to all features.

$$(NOFC > 1) \wedge (NOFC < NOF/2) \rightarrow FC = 2$$

- *Infrastructural (I)* is a class that participates in more than half of the features of a feature model.

$$(NOFC >= NOF/2) \rightarrow FC = 3$$

Feature characterizations of classes attach semantic significance to a class in terms of its role in a feature. Our feature characterization approach reduces the large feature-traces to consider only the relationships between features and software entities. Information about the frequency of references to a method or class in a feature-trace is not taken into consideration.

3 Semantic Driven Software Analysis

Common software analysis approaches focus on structural information and ignore the semantics of the problem and solution domain semantics. But this information is essential in getting a full interpretation of a software system and its meaning. As an example: the class structure of a text processor, a physical simulation or a computer game might all look the same; but the naming of the source code will differ, since each project uses its own domain specific vocabulary. *Semantic driven software analysis* gathers this information from the comments, documentation, and identifier names associated with the source code using information retrieval methods.

Our semantic analysis tool Hapax [13] uses *latent semantic indexing*, a state of the art technique in information retrieval to index, retrieve and analyze textual information

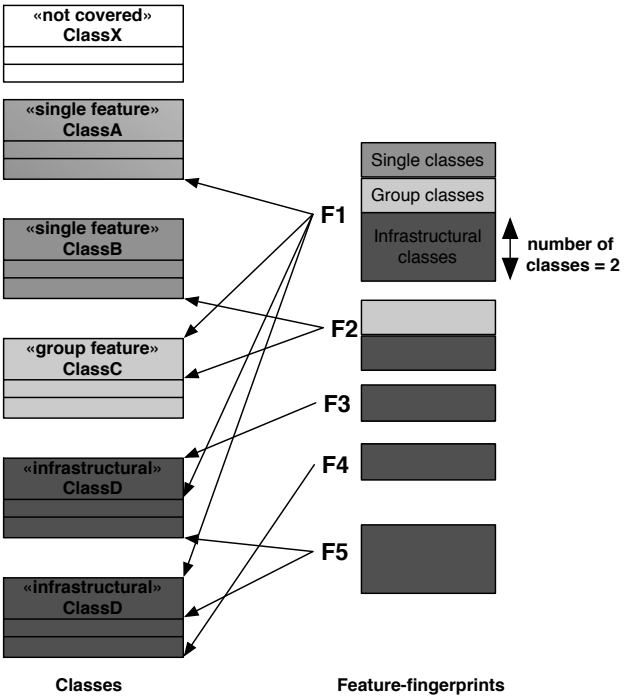

Figure 1. Feature-Fingerprints and Classes Relationships

[4]. LSI treats the software system as a set of text documents and analyzes how terms are spread over the documents. Principal components analysis is used to detect conceptual correlations and provides a similarity measurement between both documents and terms.

As most text categorization systems, LSI is based on the Vector Space Model (VSM) approach. This approach models the text corpus as a term-document matrix, which is a tabular listing of mere term frequencies. Originally LSI was developed to overcome problems with synonymy and polysemy that occurred in prior vectorial approaches. It solves this problem by replacing the full term-document matrix with an approximation. The downsizing is achieved with Singular Value Decomposition (SVD), a kind of Principal Components Analysis originally used in Signal Processing to reduce noise. The assumption is that the original term-document matrix is noisy (the aforementioned synonymy and polysemy) and the approximation is then interpreted as a noise reduced – and thus better – model of the text corpus.

Even though search engines [2] are the most common usage of LSI, there is a wide range of applications, such as: automatic essay grading [8], automatic assignment of reviewers to submitted conference papers [6], cross-language search engines [15], thesauri, spell checkers and many more. As a model, LSI has been used to simulate language

processing of the human brain, such as the language acquisition of children [14] and high-level comprehension phenomena like metaphor understanding, causal inferences and judgments of similarity.

3.1 Semantic Clustering at Work

To get a semantic model of the software system, we implemented these steps:

- First, we split the software system into text documents. While static approaches work with the source code of classes or methods, in this paper we use the textual representation of feature-traces as documents.
- The second step counts the frequencies of term occurrences in the documents. A term is any word found in the source code or comments, except keywords of the programming language. Identifiers are separated based on standard naming conventions (*e.g.*, camel-case).
- Then singular value decomposition, a principal components analysis technique, is applied on the term occurrence data. This yields an index with conceptual correlations and similarities between both documents and terms. More in-depth information on using LSI is given in [4, 2].
- To understand this semantic correlations, we group the documents using a hierarchical clustering algorithm. We visualization the clusters on a shaded correlation matrix. A shaded correlation matrix is a square matrix showing the similarity between documents in gray colors. The color at $m_{i,j}$ shows the similarity between the i-th and the j-th document: the darker the color, the more similar these two documents. The visualization algorithm itself is detailed in [13].

4 Our Approach

The novelty of our approach is the combination between dynamic analysis and semantic analysis. Our paper has two goals: to detect similarities between traces, and to detect similarities between classes based on their involvement into the traces.

We outline how we apply our technique to obtain a semantical analysis on top of feature-traces from a software system.

1. We instrument the code of the the application under analysis and execute a set of its features as described in Section 6. Our dynamic analysis tool *TraceScraper* extracts execution traces and models them as a tree of method invocation calls. We treat feature-traces as first class entities and incorporate them into the static model of the source code. By doing so we establish the relationships between the methods calls of the feature traces and the static model class and method entities. We compute the feature characterizations of the classes as described in Section 2.
2. Our semantic analysis tool *Hapax* is applied on the feature-traces. To use the feature-traces as text corpus, we create *ad-hoc* text documents with the method names found in the feature traces. Hapax applies LSI on the documents, clusters them and finally delivers a visualization of document clusters and their similarities. For more detail refer to Section 3.1.

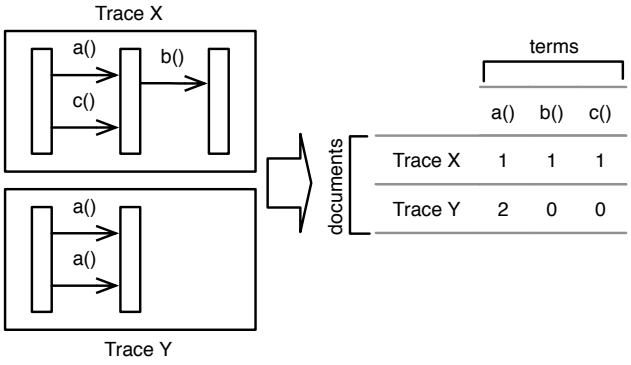

Figure 2. Example of how traces form documents and the method calls form the terms.

Both tools are built on top of our reverse engineering framework Moose [19], that provides a generic mechanism which allows for an easy composition of different tools. Because of that, we could easily integrate the two tools to perform the semantic analysis on the traces.

5 Validation: Ejp-Presenter and Smallwiki

In this section we present the results of applying our approach to the *Ejp-presenter* and the *SmallWiki* case studies.

Ejp-presenter [22] is an open source tool written in java which provides a graphical user interface for viewing execution traces of java programs. It consists of 166 classes. To obtain feature traces we instrumented 13 unit tests provided with the application. Our assumption was that each unit test exercised a distinct feature.

SmallWiki [20] is a collaborative content management system used to create, edit and manage hypertext pages on the web. It is implemented in Smaltalk and consists of 464 classes. To identify features of *SmalWiki* we associate features with the links and entry forms of the *SmallWiki* pages.

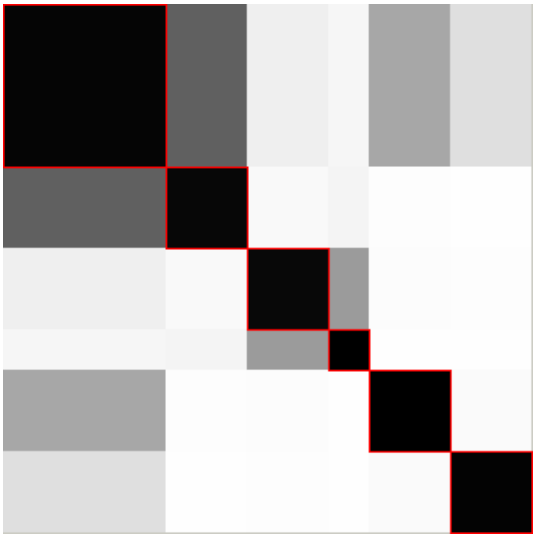

Figure 3. Correlation matrix with the features of Ejp-Presenter, showing well distributed concepts.

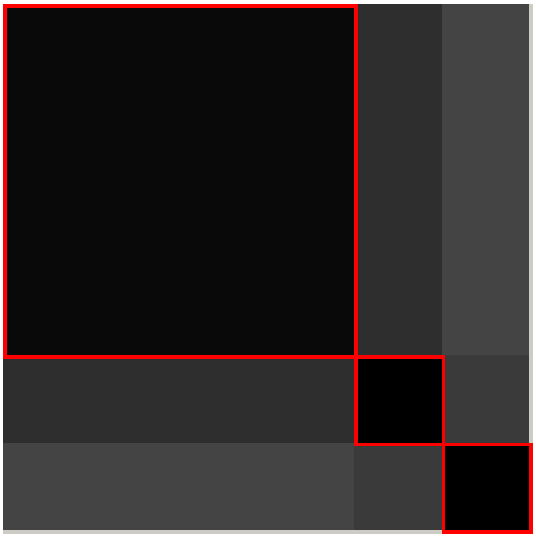

Figure 4. Correlation matrix with the features of Smallwiki, showing one concept only.

We assume that each link or button on a page triggers a distinct feature of the application. For this experiment we executed 6 features.

As mentioned in the introduction we tackle the case studies at two levels of abstraction, once using features and once using classes as granularity.

5.1 Identifying Similar Features

To identify similar features, we use the feature-traces as documents and the method calls involved in a trace as terms. Similar features are clustered together, and the clusters visualized on a shaded correlation matrix. The visualization reveals the semantic similarity between the features, showing how they are related to each other.

Figure 3 shows the *Ejp-presenter* case study. Its features are well distributed: there are 6 clusters of different sizes, and – as indicated by the gray blocks in the off-diagonal – different correlations among them.

This is a list with the features in each cluster, starting from top left to bottom right:

1. boolean parameter, string list parameter, radio parameter, and remove non significant.
2. loaded method and loaded class.
3. configuration and mainframe.
4. dom.
5. highlight hotspot and color parameter.
6. file chooser dialog and color chooser.

The names shown in the above listing are of a descriptive nature, and not part of the vocabulary used by the Information Retrieval engine itself. Thus we can judge, based on them, that the analysis revealed meaningful correlations.

Figure 4 shows the *SmallWiki* case study. Because its features use similar methods, they belong to the same semantical concept. A closer look at the feature-traces reveals that *SmallWiki* has a very generic structure, and the traces are not discriminated by their method usage but by the parameters passed to their methods. Taking only the method names into account, our approach fails discriminating these features.

5.2 Identifying Similar Classes

In Section 2 we give a characterization of classes based on their structural relationship to features. In Section 3.1 we show how we retrieve a characterization of classes based on their semantic correlation.

To identify the semantic correlation between classes, we use the classes as documents, and all method invocations originating from a class as terms. Thus classes with similar outgoing method invocations are clustered together, that is classes that are based on the same functionality belong to the same cluster. We expect these clusters to match with the 'feature terminology' characterization, since *single feature* classes are based on *group feature* classes with in turn are based on *infrastructure* classes.

Figure 5 reveals seven semantical clusters of different shape. In Table 1 we compare theses clusters – numbered

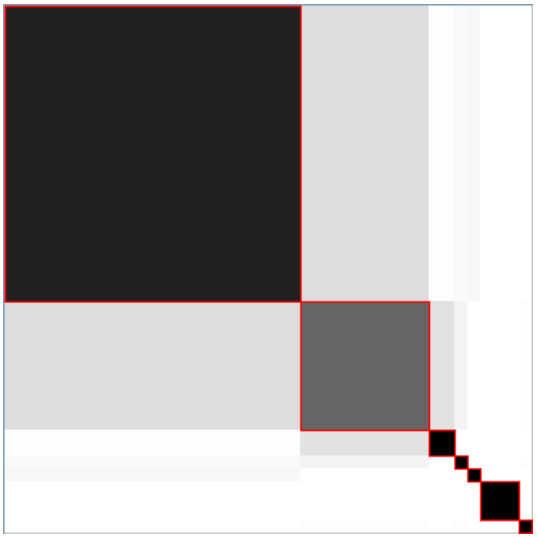

Figure 5. Correlation matrix with Ejp-Presenter classes, based on their usage in features-traces.

from top left to bottom right – with the 'feature terminology' characterization.

cluster	#1	#2	#3	#4	#5	#6	#7
single	19	4	–	–	1	3	1
group	4	6	2	–	–	–	–
infra.	–	–	–	1	–	–	–
size	23	10	2	1	1	3	1

Table 1. The clusters from Figure 5 and the types of classes contained.

And in fact, the two characterizations – once based on semantical analysis, once based on structural analysis – match pretty well.

6 Discussion

The large volume of information and complexity of dynamic information makes it hard to infer higher level of information about the system.

Coverage. We limit the scope of our investigation to focus on a set of features. Our feature model does not achieve 100% coverage of the system. For the purpose of feature location, complete coverage is not necessary. However, LSI analysis yields better results on a large text corpus. Therefore to improve our results, we need to increase the coverage the application by exercising more of its features.

Trace as Text Corpus. In this paper, we build the text corpus from the names of the methods that get called from the studied traces. When applying the approach to *SmallWiki*, the result was not very relevant because *SmallWiki* has a generic structure and the difference between traces is not given by the method names, but by the parameters passed to the methods. Hence, a variation of the approach would be to take the parameter names into consideration when building the text corpus.

Language Independence. Obtaining the traces from the running application requires code instrumentation. The means of instrumenting the application is language dependent. *Ejp-presenter* is implemented in java. To instrument it we used the *Ejp (Extensible Java Profiler)* [22] based on the Java Virtual Machine Profiler Interface (JVMPI). *SmallWiki* is implemented in Smalltalk. Our Smalltalk instrumentation is based on method wrappers [3].

We abstract a feature model from the traces we obtain by exercising the features of the instrumented system. Our analysis is performed on the feature model and is therefore language independent.

7 Related Work

Many researchers have identified the potential of feature-centric approaches in software engineering and in particular as a basis for reverse-engineering [7, 23, 24]. Our work is directly related to the field of dynamic analysis [1, 11, 25] and user-driven approaches [12].

Feature location techniques such as *Software Reconnaissence* described by Wilde and Scully [23] , and that of Eisenbarth et al. [7] are closely related to our feature location approach. In contrast, our main focus is applying feature-driven analysis to object-oriented applications.

LSI has been recently proposed in static software analysis for various tasks, such as: identification of high-level conceptual clones [17], recovering links between external documentation and source code [18], automatic categorization of software projects in open-source repositories [21] and visualization of conceptual correlations among software artifacts [13].

8 Conclusions and Future Work

Reverse engineering approaches that focus only on the implementation details and static structure of a system overlook the dynamic relationships between the different parts that only appear at runtime. Our approach is to complement the static and dynamic analysis by building a model in which features are related to the structural entities.

Dynamic analysis offers a wealth of information, but it is exactly the wealth of information that poses the problem in the analysis. To deal with it, we employed Latent Semantic Indexing, an information retrieval technique that works

with documents and terms . Our goals were to identify related features and to identify related classes that participate in features. We use the method calls from the traces as the text corpus and then we use two mappings to documents: (1) traces as documents, and (2) classes as documents. We clustered the documents based on the terms used to find relationships between them.

The results obtained on two case studies are promising, yet we did encounter problems with only considering the method names as text corpus. From our findings we conclude that more work is needed to assess the different variations of the approach. Furthermore, LSI yields better results on large text corpus, hence we also need to apply our approach on larger case studies or to achieve a higher coverage of the system by our feature-traces.

Acknowledgments: We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "RECAST: Evolution of Object-Oriented Applications" (SNF Project No. 620-066077).

References

[1] T. Ball. The Concept of Dynamic Analysis. In *Proceedings of ESEC/FSE '99 (7th European Software Engineering Conference and 7th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, number 1687 in LNCS, pages 216–234, sep 1999.

[2] M. W. Berry, S. T. Dumais, and G. W. O'Brien. Using linear algebra for intelligent information retrieval. Technical Report UT-CS-94-270, 1994.

[3] J. Brant, B. Foote, R. Johnson, and D. Roberts. Wrappers to the Rescue. In *Proceedings ECOOP '98*, volume 1445 of *LNCS*, pages 396–417. Springer-Verlag, 1998.

[4] S. C. Deerwester, S. T. Dumais, T. K. Landauer, G. W. Furnas, and R. A. Harshman. Indexing by latent semantic analysis. *Journal of the American Society of Information Science*, 41(6):391–407, 1990.

[5] S. Demeyer, S. Ducasse, and O. Nierstrasz. Finding refactorings via change metrics. In *Proceedings of OOPSLA '2000 (International Conference on Object-Oriented Programming Systems, Languages and Applications)*, pages 166–178, 2000.

[6] S. T. Dumais and J. Nielsen. Automating the assignment of submitted manuscripts to reviewers. In *Research and Development in Information Retrieval*, pages 233–244, 1992.

[7] T. Eisenbarth, R. Koschke, and D. Simon. Locating Features in Source Code. *IEEE Computer*, 29(3):210–224, Mar. 2003.

[8] P. W. Foltz, D. Laham, and T. K. Landauer. Automated essay scoring: Applications to educational technology. In *Proceedings of EdMedia '99*, 1999.

[9] O. Greevy and S. Ducasse. Correlating features and code using a compact two-sided trace analysis approach. In *Proceedings of CSMR 2005 (9th European Conference on Software Maintenance and Reengineering*, pages 314–323. IEEE Computer Society Press, 2005.

[10] O. Greevy, S. Ducasse, and T. Gîrba. Analyzing feature traces to incorporate the semantics of change in software evolution analysis. In *Proceedings of ICSM 2005 (21th International Conference on Software Maintenance)*, pages 347–356. IEEE Computer Society Press, Sept. 2005.

[11] A. Hamou-Lhadj, E. Braun, D. Amyot, and T. Lethbridge. Recovering behavioral design models from execution traces. In *Proceedings of CSMR 2005 (9th European Conference on Software Maintenance and Reengineering*. IEEE Computer Society Press, 2005.

[12] I. Jacobson. Use cases and aspects—working seamlessly together. *Journal of Object Technology*, 2(4):7–28, July 2003.

[13] A. Kuhn, S. Ducasse, and T. Gîrba. Enriching reverse engineering with semantic clustering. In *Proceedings of Working Conference On Reverse Engineering (WCRE 2005)*, pages ??–??, Nov. 2005. to appear.

[14] T. Landauer and S. Dumais. The latent semantic analysis theory of acquisition, induction, and representation of knowledge. In *Psychological Review*, volume 104/2, pages 211–240, 1991.

[15] T. Landauer and M. Littmann. Fully automatic cross-language document retrieval using latent semantic indexing. In *In Proceedings of the 6th Conference of the UW Centre for the New Oxford English Dictionary and Text Research*, pages 31–38, 1990.

[16] M. Lanza and S. Ducasse. A Categorization of Classes based on the Visualization of their Internal Structure: the Class Blueprint. In *Proceedings of OOPSLA '01 (International Conference on Object-Oriented Programming Systems, Languages and Applications)*, pages 300–311. ACM Press, 2001.

[17] J. I. Maletic and A. Marcus. Supporting program comprehension using semantic and structural information. In *Proceedings of the International Conference on Software Engineering (ICSE 2001)*, pages 103–112, 2001.

[18] A. Marcus and J. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing, 2003.

[19] O. Nierstrasz, S. Ducasse, and T. Girba. The story of Moose: an agile reengineering environment. In *Proceedings of ESEC/FSE 2005*, pages 1–10. ACM, 2005. Invited paper.

[20] L. Renggli. Smallwiki: Collaborative content management. Informatikprojekt, University of Bern, 2003.

[21] M. M. Shinji Kawaguchi, Pankaj K. Garg and K. Inoue. Mudablue: An automatic categorization system for open source repositories. In *Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC.04)*, 2004.

[22] S. Vauclair. Extensible java profiler. Masters thesis, EPF Lausanne, 2003.

[23] N. Wilde and M. C. Scully. Software reconnaisance: Mapping program features to code. *Software Maintenance: Research and Practice*, 7(1):49–62, 1995.

[24] W. E. Wong, S. S. Gokhale, and J. R. Horgan. Quantifying the closeness between program components and features. *J. Syst. Softw.*, 54(2):87–98, 2000.

[25] A. Zaidman, T. Calders, S. Demeyer, and J. Paredaens. Applying webmining techniques to execution traces to support the program comprehension process. In *Proceedings of CSMR 2005 (9th European Conference on Software Maintenance and Reengineering*. IEEE Computer Society Press, 2005.