The Feature Signatures of Evolving Programs *

Daniel R. Licata, Christopher D. Harris and Shriram Krishnamurthi
Computer Science Department
Brown University

1. Introduction

As programs evolve, their code increasingly becomes tangled by programmers and requirements. This mosaic quality complicates program comprehension and maintenance. Many of these activities can benefit from viewing the program as a collection of *features*. We introduce an inexpensive and easily comprehensible summary of program changes called the *feature signature* and investigate its properties. We find a remarkable similarity in the nature of feature signatures across multiple non-trivial programs, developers and magnitudes of changes. This indicates that feature signatures are a meaningful notion worth studying. We then show numerous applications of feature signatures to software evolution, establishing their utility. A more comprehensive version of this paper (including additional case study results) is also available [16].

Building programmer-friendly tools means both leaving the development process intact and limiting the amount of extra work that the programmer must do, relying instead on software artifacts that are kept up-to-date as the software evolves. One such artifact is the program itself; however, any ascription of human knowledge requires some form of redundant specification beyond the program source. The most natural place to look for redundancy is in design documents and other forms of documentation. Sadly, any portion of the software suite that is not immediately useful to developers and that suffers from poor tool support tends to be neglected; documentation is notorious in this regard. Fortunately, there is one source of redundancy that programmers often maintain, because of its utility: the *test suite*.

2. Test Suites and Features

In this paper, we will use the term *test case* for the input/output pair necessary to complete a single execution of a program and specify the expected result; *test suite* for a collection of test cases; and *test battery* for a program's complete set of test suites. A feature is a product characteristic that customers find important in describing and distinguishing related software systems. We make a crucial assumption about the structure of test suites: we assume that *the test battery is partitioned into suites that are roughly aligned with the features of the system*.

Why should tests align with features at all? In many cases, testing is conducted by people outside the development process; these testers can view the system *only* in terms of its features (which they derive from the requirements documentation), not its implementation. Even when developers and testers coincide, tests typically measure the input-output behaviors of a program—which correspond to the features that a user sees. When individual test cases themselves correspond to some small part of the functionality of a program that the user can see, it is easy to collect them into suites based on the features that those bits of functionality comprise.

3. Analysis Methodology

We begin by assuming that we have two versions of the program. This is usually easy to reconstruct from a standard version control system such as CVS [2]. For simplicity, we also assume that the test battery does not change between the two states of the system.

Given these inputs, the methodology for extracting data about the program from the test battery is simple:

1. Use a differencing utility which consumes the two program sources and generates a list of blocks of code that have been added, deleted, or changed between the two versions.

2. Run the test battery on both versions of the program, gathering profiling information (the frequency of execution of each block of code) for each test suite. The profiling tool needs to monitor execution at the same level of granularity captured by the differencing tool.

The data that ensue from this process have the following form:

*Research partially supported by NSF grants ESI-0010064, ITR-0218973, SEL-0305949 and SEL-0305950, and by Brown University's Karen T. Romer UTRA program.

	ts 1	ts 2	ts 3	⋯	ts m
block 1	1	0	0		0
block 2	1	1	0		0
⋯					
block n	0	0	1		0

Each row of this table corresponds to one block of code that the differencing utility identified as being added, deleted, or changed in this modification to the program. Each column depicts the execution of one test suite. A 1 in a particular entry means that that particular code block was exercised (as recognized by the profiler) by that test suite; a 0 means that it was not. For additions and changes, we use the profiling results from the program after the change; for deletions, we must use the results from the program before the change. We call each row that difference block's *feature signature*: the feature signature of a block of code is the vector of 0's and 1's that indicates which test suites executed it.

Having obtained the feature signature of each difference block, we can then summarize these data into a simple graphical form. First, we compute the number of 1's in the feature signature of each difference block, which tells us how many test suites impacted that modification. We then generate a histogram of these counts.

4. Parameters for Case Study

This paper contains a case study that analyzes the form of feature signatures. Our study employed two large software systems, each with a significant number of features: the standard interpreter for the Python [3] language, and MzScheme [11], the virtual machine for the DrScheme programming environment [10]. Though both are language implementations, their implementation details and development methodologies differ in enough ways to mask superficial similarities. These systems are Open Source, so our experiments are easily repeatable.

The tests we used for MzScheme were factored into 25 to 38 suites, while the tests for Python were factored into 113 to 196 suites; the tests for these programs were all manually written and came factored by feature.

Both of these programs are written in C, so we chose to adopt the naïve differencing that Unix's `diff` offers and to profile with `gcov` (part of the `gcc` compiler suite), which reports profiling information for each executable line. In our reports, we adopt the convention that a block executes if at least one of the lines in it executes (according to `gcov`).

5. Feature Signature Analysis

Figure 1 is an example of the histograms that result from our case study (the full set of histograms is also available [16]). The ith bar of the histogram tells the number of

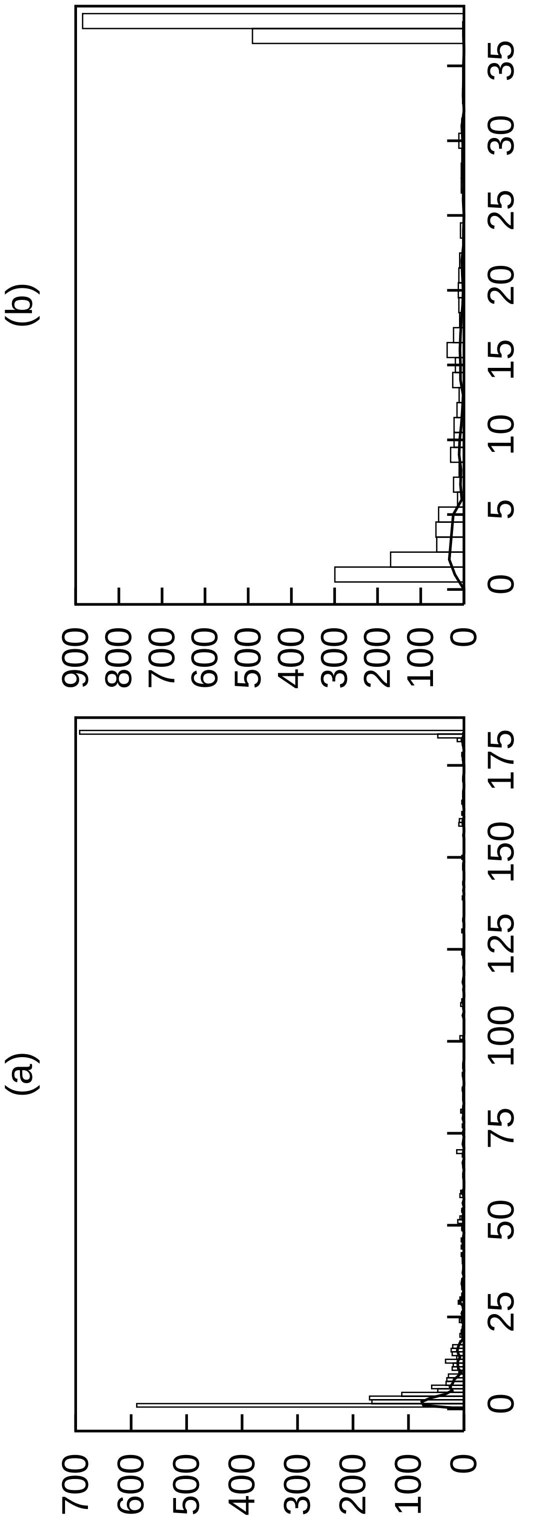

Figure 1. Feature signature histograms
Histograms of the number of difference blocks executed by exactly each number of different test suites for changes to (a) Python and (b) MzScheme

changed blocks of code that were executed by exactly i test suites. The left-hand-side of each graph thus shows the number of change blocks that were executed by only a small number of test suites, while the right-hand-side shows how many were impacted by almost all or all of the test suites.

We created these histograms for many small (27 to 56 diff blocks) changes to Python, each on the granularity of between one and five CVS commits. Examining these graphs showed that they have a very distinctive shape: most differences blocks fall near the left or right edge of the histogram. Very few of the blocks fall in the middle. We found that these graphs were all *nearly identical in shape*, despite being for semantically unrelated changes to the program.

Figure 1(a) presents a much larger change to Python—at the level of significant release (between versions 1.5.2 and 2.2, a change affecting 2442 difference blocks in several hundred commits). Remarkably, we found that this graph (and that of another change to Python of the same magnitude) had the same shape as those for the small changes—change blocks concentrated at the left and right ends.

Finally, we found the same histogram shape in our experiments on the MzScheme code base. Figure 1(b) shows the histogram for the change between versions 103 and 200, a change taking 21 months and affecting 2400 difference blocks. Another change to MzScheme involving about 700 differences exhibited the same shape as well.

We have manually studied the difference blocks in many of these cases, and in each case found that the number of suites impacting the block is consistent with the changed block's actual impact:

1. Blocks that are on the right appear to affect all uses of the system. In the case of Python and MzScheme, we find that these tend to be changes to infrastructure such as the garbage collector. We therefore call such differences *infrastructural*.

2. Blocks that are on the left appear to pertain to a very small number of features. We label these *featuristic* .

The shape of the typical histogram says that *almost all change blocks are either featuristic or infrastructural*. The remarkable similarity in shape across both different software systems and different magnitudes of edits suggests that a technique based upon this property of feature signatures may apply broadly.

6. Clustering

While feature signatures give us some information about program changes, for many of the applications we describe later the number of differences is simply too many. Obviously, a programmer cannot contend with thousands of little difference blocks; we must group these into clusters of *conceptual changes*—aggregates of changed blocks that the programmer thinks of as one change to the program. In the programs we studied, there was always more than one conceptual change per CVS commit (if there were only one, then we could simply make each commit a single cluster).

Since programmers often identify conceptual changes by the program features they impact, clustering using feature signatures is likely to be effective. One easy clustering technique is to simply group together all of the code blocks with identical feature signatures. We call the clusters that ensue from this grouping *naïve clusters*.

To analyze this technique's utility, we compared the conceptual changes identified in the (remarkably detailed) CVS change logs for seven small changes to Python with our naïve clustering. In our study of the naïve clusters, all the code blocks grouped into a naïve cluster were clearly identifiable as part of the same conceptual change to the program. In addition, the test suites that exercised the edits in a cluster corresponded with the features that the edits impacted (according to the change logs). Furthermore, our analysis showed that, for both small and large edits, the number of naïve clusters was significantly less than the number of change blocks. (The line graphs in Figure 1 denote the number of naïve clusters.)

While the naïve clusters are a significant improvement over looking at blocks individually, not all change blocks that are part of the same conceptual edit always fall within the same cluster. Advances in clustering techniques can thus improve our results; this remains a significant area for future research.

7. Applications

Feature signatures and their clusters are versatile: they give rise to numerous useful and diverse analyses.

7.1. Code Rationale Construction

Any programmer who has worked on an unfamiliar software system is accustomed to looking at a baffling piece of code and trying to piece together a guess about that code's role in the larger system. They would benefit greatly from having a *rationale* for the code in question. This rationale is, unfortunately, rarely documented well. Programmers sometimes have difficulty justifying their own code, so manually reconstructing a rationale for someone else's program is daunting. The problem compounds when the original programmer who wrote the code is not easily accessible, or at any rate no longer has a stake in the project (which is especially typical of many Open Source projects).

There are two ways to apply the information from feature signatures to the rationale problem. The first is a tool that helps programmers write better rationales when they perform a commit: the feature signature can be used to create templates of rationale logs that are then kept in version control software. Given a cluster of blocks that are part of a conceptual edit, the tool can supply the union of their non-infrastructural feature signatures as the rationale template. The author of the code is presented with a list of the features that each of his changes impacted. By giving much finer-grained information than merely which files changed, these templates prompt programmers to provide meaningful descriptions of changes and remind them of changes they may otherwise forget to document. They also can help a subsequent code browser identify incomplete change logs. Furthermore, by being lightweight and automatic, this process is easy to integrate with a tool such as CVS.

The second use is a related tool that helps later programmers discover rationales ex post facto for poorly-annotated changes. Given an unclear code fragment, the programmer gets the history of changes that impacted the code, then applies our methodology to pairs of versions; the features that the code impacted at each change then tell the programmer something about why the code evolved to its current state.

The key reason that these techniques are meaningful is the pattern of featuristic and infrastructural changes: a code block will likely either have only a few features in its rationale or be infrastructural. Rationale generation would also benefit from detailed knowledge of which individual test *cases* impacted the difference, especially if the number of test cases is small. We have not explored this extension in the present work.

7.2. Test Suite Structure Investigation

Following the work of Birkhoff [6], Ganter and Wille [12] describe *concept analysis* as a way of understanding and clustering data. Concept analysis lets users alternate between tabular and hierarchical views of lattices. In particular, given the tabular view, it lets the user construct a lattice whose ordering relationship identifies the relationship between maximal collections of "objects" (in our case, difference blocks) that have the same set of "attributes" (here, test suites).

We can apply concept analysis to our feature signatures. Studying the concept lattice helps the user understand potentially subtle relationships between the test suites. For instance, she could identify test suites with the relationship that whenever the first exercised a difference block, so did the second. We can also use this lattice to improve the clustering of difference blocks as well. As Section 6 explains, while naïve clusters are useful, they sometimes draw too many distinctions. Relaxing this clustering amounts to knowing when a 0 in one feature signature can "match" a 1 in another. The relationships between test suites that concept analysis identifies can help determine when such matching is appropriate.

7.3. Associating Related Changes

When developers commit significant changes to a codebase, they often add important new features to the system. Yet each new feature often corresponds to edits to multiple non-contiguous portions of the program source. (Our finding multiple difference blocks that should be clustered into a single feature supports this idea.)

Many code editors and browsers use colors to highlight syntactic "parts of speech"; however, this provides little information about the program's semantics. Visually identifying features is likely to be much more useful to a user. Given the clustering of edits by feature signature, it is easy to color each feature differently; syntactically distant but semantically related edits are then associated by a color corresponding to the feature set they affect.

In principle, a programmer should then be able to take these bits of code implementing a single feature and isolate them into a module; this is the intuition behind aspect-oriented programming [14]. Unfortunately, this process runs into shortcomings in modern aspect technology. Some forms, such as AspectJ [13], are very good at performing an intrusive (i.e., without respect to modular boundaries and interfaces) but consistent change at many places in the program. Other forms, such as mixin layers [21], are best at performing disparate changes but only at well-defined points. The changes we identify are both intrusive and disparate. This suggests the need for better aspect technology to capture and modularize the kinds of changes we notice occurring in practice.

8. Related Work

Many aspects of our work rely on techniques that have been thoroughly treated in the literature. Clustering is one such technique; Fasulo [9] provides an overview of current clustering techniques. Concept analysis was pioneered by Ganter and Wille [12], and has since been applied to program comprehension and refactoring [20, 22].

Our work is closely related to dynamic slicing [15] and similar tracing techniques. This has mainly been employed for program comprehension and debugging [15] and visualizations [4, 17]. For instance, χSuds [1] lets the user visualize code features using coloring schemes similar to the one we propose, but lacking clustering; we believe the volume of data would pose an overwhelming cognitive burden to the programmer.

In the same vein, Mehta and Heineman [18] describe a process for refactoring legacy systems into components; however, they do not describe and exploit patterns of software evolution. They offer a technique for clustering test cases into feature-specific suites, which we can exploit when applying our work to systems whose test batteries are not already so factored. Wilde and colleagues [23] describe a dynamic analysis based on test cases for discovering the code implementing only a single specific feature. Eisenbarth, Koschke and Simon [8] expand this work to multiple features using concept analysis, but they use this analysis only to assist the programmer in manual feature discovery.

Baxter [5] captures code rationale as the goals met by program transformations, but only for those who program by iterative transformations of a formal specification. Egyed [7] uses a profiling technique to help generate and validate associations among source code and other software artifacts; code rationale could then be presented in terms of those associations. Other program comprehension methods help programmers understand the overall design of a system (rather than a specific bit of code); for instance, Murphy and colleagues [19] provide a method for finding where an abstract model of a system and the source code diverge.

9. Conclusion and Future Work

We have presented a lightweight yet effective technique for studying the evolution of programs. We propose a notion of *feature signature*, which identifies the features of a system that impact a change. We determine impact dynamically by profiling using a program's test suites.

Our experiments with two significant software systems shows that feature signatures are a useful measure of changes. In particular, we find that most changes tend to pertain to either a very small number of features or to almost all of them. As a result, we can draw a distinction between changes made to the program's infrastructure and ones made to implement or modify very specific features. We also use the feature signatures as inputs to clustering algorithms to group related changes. We then show that the bimodal nature of changes, and the availability of clusters, lead to numerous useful applications. We present prototypes of tool support for most of these applications.

For future work, we need to consider many more sources of information, such as fine-grained changes to test suites, success and failure of test cases, profiling counts, better differencing techniques, and better clustering algorithms (such as ones that can exploit the test suite relationships described by the concept lattice). Additionally, experiments with other software systems would test our claims about the shape of feature signatures.

References

[1] χsuds manual. http://xsuds.argreenhouse.com/.
[2] `cvs` user's guide. http://www.cvshome.org/.
[3] Python language. http://www.python.org.
[4] T. Ball. Software visualization in the large. *IEEE Computer*, 29(4):33–43, April 1996.
[5] I. Baxter. Design maintenance systems. *Communications of the ACM*, Vol. 4, April 1992.
[6] G. Birkhoff. Lattice theory. *American Mathematical Society Colloquium Publications*, 25, 1967.
[7] A. Egyed. A scenario-driven approach to trace dependency analysis. *Transactions on Software Engineering*, 29(2), 2003.
[8] T. Eisenbarth, R. Koschke, and D. Simon. Aiding program comprehension by static and dynamic feature analysis. In *Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society Press, November 2001.
[9] D. Fasulo. An analysis of recent work on clustering algorithms. Technical Report 01-03-02, University Of Washington, 1999.
[10] R. B. Findler, J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler, and M. Felleisen. DrScheme: A programming environment for Scheme. *Journal of Functional Programming*, 12(2):159–182, 2002.
[11] M. Flatt. PLT MzScheme: Language manual. Technical Report TR97-280, Rice University, 1997.
[12] B. Ganter and R. Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer Verlag, 1999.
[13] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In *European Conference on Object-Oriented Programming*, 2001.
[14] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming*, June 1997.
[15] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, 1998.
[16] D. R. Licata, C. D. Harris, and S. Krishnamurthi. The feature signatures of evolving programs. Technical Report CS-03-12, Brown University Department of Computer Science, 2003.
[17] A. Malony, D. Hammerslag, and D. Jabalonski. Traceview: A trace visualization tool. *IEEE Software*, pages 19–28, September 1991.
[18] A. Mehta and G. T. Heineman. Evolving legacy system features into fine-grained components. In *Proceedings of the 24th International Conference on Software Engineering*, pages 417–427. ACM Press, 2002.
[19] G. C. Murphy, D. Notkin, and K. J. Sullivan. Software reflexion models: bridging the gap between design and implementation. *Transactions on Software Engineering*, 27(4), 2001.
[20] M. Siff and T. Reps. Identifying modules via concept analysis. In *International Conference on Software Maintenance*, pages 170–179. IEEE Computer Society Press, 1997.
[21] Y. Smaragdakis and D. Batory. Implementing layered designs and mixin layers. In *European Conference on Object-Oriented Programming*, pages 550–570, July 1998.
[22] G. Snelting and F. Tip. Understanding class hierarchies using concept analysis. *ACM Transactions on Programming Languages and Systems*, 22:540–582, May 2000.
[23] N. Wilde and C. Casey. Early field experience with the software reconnaissance technique for program comprehension. In *International Conference on Software Maintenance*. IEEE Computer Society Press, 1996.