The Feature Signatures of Evolving Programs [*]

Daniel R. Licata, Christopher D. Harris and Shriram Krishnamurthi
Computer Science Department
Brown University
Contact: dlicata@cs.brown.edu

Abstract

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, establishing their utility.*

1 Introduction

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. Constructing this rationale involves mining the history of the program's development to understand how and why it entered the system.

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. This is especially typical of many Open Source projects, which are developed worldwide.

This situation presents a challenge and an opportunity for building tools that assist programmers with rationales. An effective tool would be useful both to programmers who commit changes to a codebase (by helping them endow it with better documentation) and to those who have to study changes in retrospect.

Other researchers have proposed tools and methodologies for automatically ascribing rationales during software development. Some of their solutions, which we describe in Section 9, call for severe changes in the way the programmer works. While such changes might feasibly be imposed on closed, strongly hierarchical groups of programmers, they may be unrealistic goals in, for example, the global development environments that are typical of Open Source software. In such environments, the set of developers is essentially unrestricted. Moreover, to the extent that there are central managers, their impositions on contributors have to be minimal: adopting a large-scale, centralized, top-down process is likely to drive away the volunteer programmers who are the lifeblood of such projects. In short, the tools must match the constraints of the domain rather than trying to force the domain to meet the operating specifications of the tools.

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: its static code, its dynamic behavior, or both; after all, it is the program that results in the executable that consumers want. However, any ascription of human knowledge requires some form of redundant specification beyond the program source itself.

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*. We therefore turn to test suites in this work.

This work exploits one other attribute of modern soft-

[*]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.

ware systems: the ability to travel in time to the point when a line of code enters the program, and reconstruct the program at that time. This is especially true of Internet-scale Open Source systems, which thrive by providing regular snapshots of the entire development state. Our experiments are therefore based on two such systems.

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. In most of what follows, we will focus on the level of test suites.

Our work is fairly sensitive to the quality of test suites. We believe there are several reasons why programmers tend to maintain test suites at least as well as any non-source artifacts:

- Unlike many design notations or documentation, test suites are *executable*. Therefore, it is easy to measure their conformance against the program (or vice versa) and keep them up-to-date.

- They provide immediate feedback by reporting errors, rather than offering more vague measures of validity. Therefore, their value is more immediately apparent than that of, say, formal specifications or documentation.

- They provide high return-on-investment, since test cases usually survive for a long time (and the mere act of changing a test case's output is useful for a programmer to know).

- As software construction becomes more distributed, developers need automated means to ensure a measure of correctness, and testing is the most readily available technique.

We make a crucial assumption about the structure of test suites in a battery: we assume that *the test battery is partitioned into suites that are roughly aligned with the features of the system*.[1]

What exactly is a feature, and how does it differ from an object? A feature is a product characteristic that customers find important in describing and distinguishing related software systems. Call forwarding in telephony systems and alarm notification in emergency service radios are but two examples. Features have the distinguishing characteristic that their implementation impacts (or "cross-cuts") objects in a system, which tends to scatter their code throughout a system's code base. Specifying and building systems explicitly as compositions of features aligns software structure with users' requirements, and is thus a powerful corrective to this scattering of code and concerns. A product line, in particular, views software as a collection of features that can be composed to create individual products [11].

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. In addition, tests typically measure the input-output behavior of a program, and the only externally viewable behaviors should be in terms of the requirements (often expressed through use cases). 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.

The assumption of tests aligning with features is present in other experimental work, such as that of Mehta and Heineman [31]. We provide further empirical evidence in Section 5. We have found that in the systems we have studied for this paper (and others we have examined informally, or even developed ourselves), even when testers and developers coincide, most test suites still decompose largely by feature.

As an example of factoring by feature, consider the test battery of a programming language interpreter. The test cases typically consist of sample programs that get run through the interpreter, paired with their expected values. Natural groups arise from these cases. For instance, there will be some collection of tests that deal with numerical primitives, some that deal with manipulating character strings, some that deal with the scoping of procedures, and so on. These groups correspond to the *linguistic features* that the language provides to its users. Sometimes, the suites are even organized by the sections of the programming language's reference manual!

Describing program changes in terms of features is particularly useful to a developer new to a system. New developers typically have only a spotty understanding of a program's structure; instead, they will run the program and first form their model of it through its user-observable features. Therefore, the test suites subscribe a vocabulary that roughly corresponds to the user's, and thus new developer's, ontology of a program.

The systems we studied came with good manually constructed test batteries, so these are the only kind we have studied. Many researchers have considered automatically generating test suites from specifications [35] and other sources. We believe having such batteries would not interfere with the application of our technique; if anything, test

[1]We recognize that some test suites exist for testing features, while programmers devise others to capture the internal behavior of the program. These are usually quite easy to distinguish.

suites generated from specifications are even more likely to conform closely to features.

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 instance, a programmer trying to understand a specific line of code may use the version controller to first determine when that code entered the codebase, then to reconstruct the state of the system just before and immediately after the change that committed it.

For simplicity, we also assume that the test battery does not change between the two states of the system. While it is usual for test batteries to grow, they usually do not undergo a massive restructuring between commits, so this assumption is quite reasonable.

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

1. Use a differencing utility to contrast the two versions of the program. The utility needs to consume the two program sources and generate a list of blocks of code that have been added, deleted, or changed between the two versions. The output may need processing so that contiguous differences get coalesced into single blocks, in the style of the Unix `diff` utility.

2. Run the test battery on both versions of the program, gathering profiling information at the level of individual test suites. Profiling tools are available for most languages; they essentially instrument the execution of a program to record the frequency of executing each unit of code. The profiling tool needs to monitor execution at the same level of granularity being captured by the differencing tool: that is, if the differencer considers individual lines of code, the profiler must also track each line, not just each function.

The data collection procedure should ideally be both lightweight (in terms of computational effort) and non-intrusive (in terms of human effort).

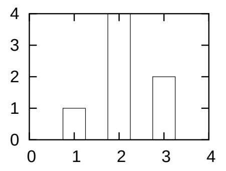
The data that ensue from this process have the following form:

	ts 1	**ts 2**	**ts 3**	$\cdots$	**ts** m
block 1	1	0	0		0
block 2	1	1	0		0
block 3	0	0	0		0
block 4	1	1	1		1
$\vdots$					
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 run. 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.

Each row of this table tells us how the test suites impacted the edit block on that row. In turn, the test suites tell a story of the features that the edit likely impacts: the row provides a capsule summary of the edit's relationship to the system's features. We therefore 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 sum of 0's and 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 sums. This results in a précis of the ways in which the test suites exercised the changes:



This is a histogram of the count of difference blocks whose feature signatures have as many 1's as the corresponding x-axis value, that is, the count of difference blocks that were executed by exactly x different test suites.

At this point, we ask the reader to pause and consider what shape they expect of this histogram.

Where did the 0's go? The column of 0's in the histogram counts the number of difference blocks that *were not exercised by any test suites*. We should be dismayed to find any differences fall in this category! Yet in fact, we do find several such differences in our experiments, but most of these are harmless because they fall into two acceptable categories:

1. The difference is caused by a comment or other metadata, not changes to the executing source.

2. The changes reside under pre-processor directives that don't get activated in this particular build. This is particularly common in programs like virtual machines that contain directives to parameterize the program by platform.

We therefore left these out of the histogram.

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:

- The standard interpreter for the Python [3] language, written by Guido van Rossum and others.
- MzScheme [17], the virtual machine for the DrScheme programming environment [16] and other applications.

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. (We list ranges because the number of tests changes over time.)

Though both are language implementations, their implementation details differ in enough ways to mask superficial similarities. In addition to significant differences in the languages they implement (some of whose features affect the rest of the language and significantly alter implementation strategy), the internals are structured as an interpreter versus a partial compiler, employ a conservative garbage collector [10] versus a reference counter, support different numbers of platforms, etc. In addition, they have different development models, release schedules and numbers and types of contributors.

We chose these language implementations for four reasons. First, they represent non-trivial implementation efforts over many years. Second, they have a significant number of features (in this case, largely the language constructs; though Scheme itself [23] is a small language, MzScheme supports exceptions, classes, mixins, two module systems, and more). Third, by already being familiar with the general architecture of programming language implementations, we were able to save the time of acquainting ourselves with the ontologies of the systems, which we would need to know to manually evaluate the success of our methodologies. Finally, we use these freely available Open Source programs to make it easy for others to repeat our experiments.

We emphasize that we worked with both systems "in a state of nature". That is, starting from the publicly available implementations, we made no changes to source code, and only minor changes to test batteries. The changes to the batteries primarily involved

- disabling test suites incompatible with our hardware (e.g. audio tests), and
- separating multiple, clearly-identified suites that resided in a single file into separate files.

We should stress that we were careful to not exploit the third author's developer status in the DrScheme project; we treated the MzScheme corpus exactly as we did that of Python, working strictly through the anonymous CVS access provided to all.

Both of these programs are written in C. We needed to choose corresponding tools for differencing and profiling. We chose to adopt the naïve differencing that Unix's `diff` offers. Because `diff` reports differences as contiguous blocks of changed lines, the function-level profiling of `gprof` was inappropriate; we therefore instead used `gcov`, which is part of the `gcc` compiler suite. In our reports, we adopt the convention that a block executes if at least one of the lines in it executes (according to `gcov`); difference blocks are not restricted to being basic blocks [4]. Fortunately, for manually-written code in C, the line is a useful unit of abstraction.

Note that this data collection procedure meets our original goal of being both lightweight and non-intrusive. It is (hopefully!) common practice to run a test battery before committing changes. Our system can easily run in the background to collect its information, since it requires no additional programmer intervention.

5 Feature Signature Analysis

Figures 1 and 2 show the histograms that result from our case study. Recall that the ith bar of the histogram tells the number of 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 most of the test suites.

Figure 1 presents histograms for the following (relatively small) changes to Python:

(a) The total number of difference blocks is 42. The change allows Python users to subclass builtin classes.

(b) The total number of difference blocks is 56. The change adds deep object comparison to the language.

(c) The total number of difference blocks is 32. The change modifies the default behavior of the division operator.

(d) The total number of difference blocks is 27. The change reflects an optimization of the unary arithmetic operators.

Each of these changes was made over the course of between one and five CVS commits. Examining these graphs shows 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. Note that the above changes are semantically unrelated, so the similarity in feature signature structure is not (as best as we can tell) in any way due to a deep similarity in the language itself or the implementation of these particular operations.

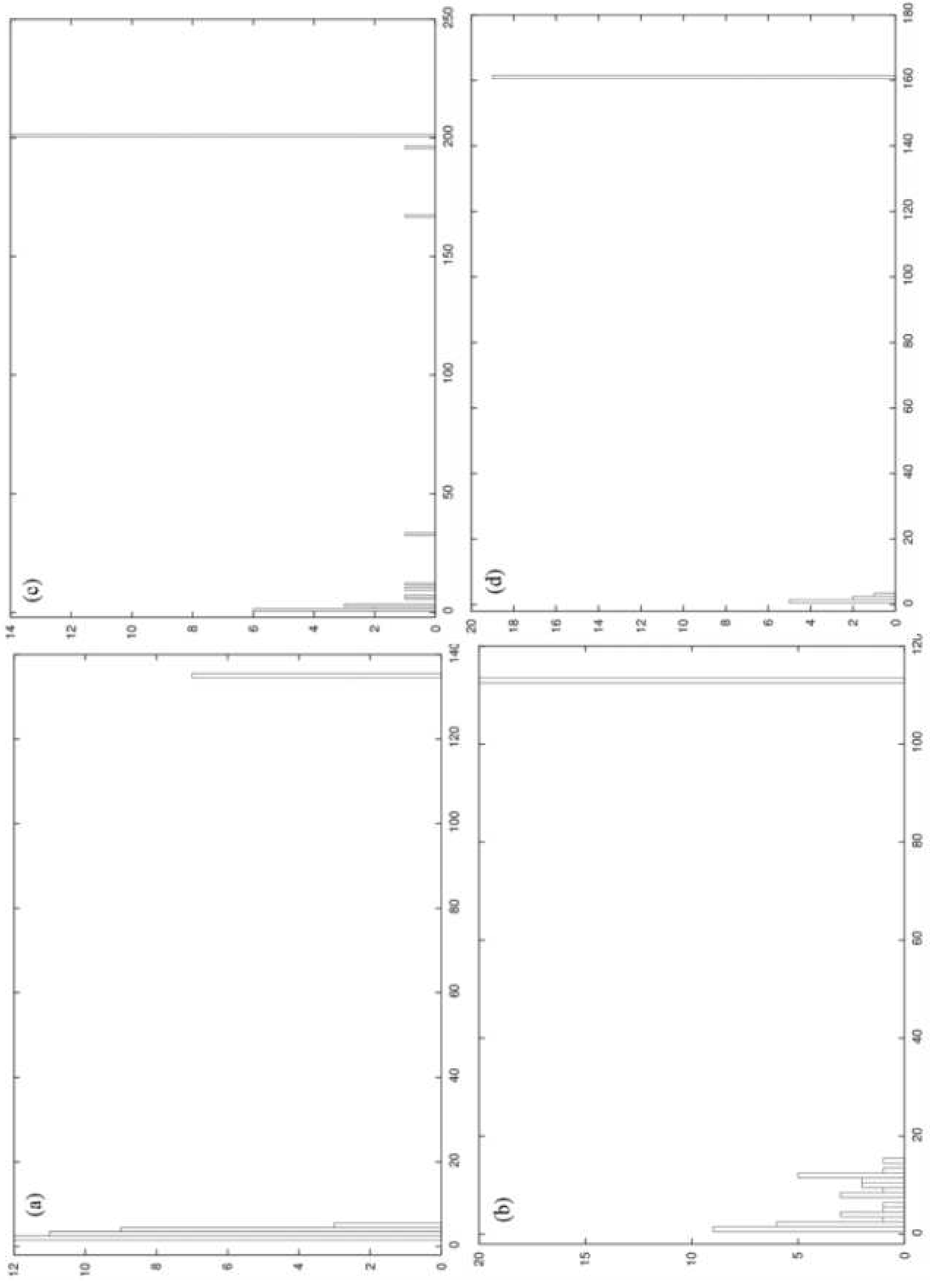


Figure 1. Feature signature histograms for small changes to Python

These are histograms of the number of difference blocks that were executed by exactly each number of different test suites.

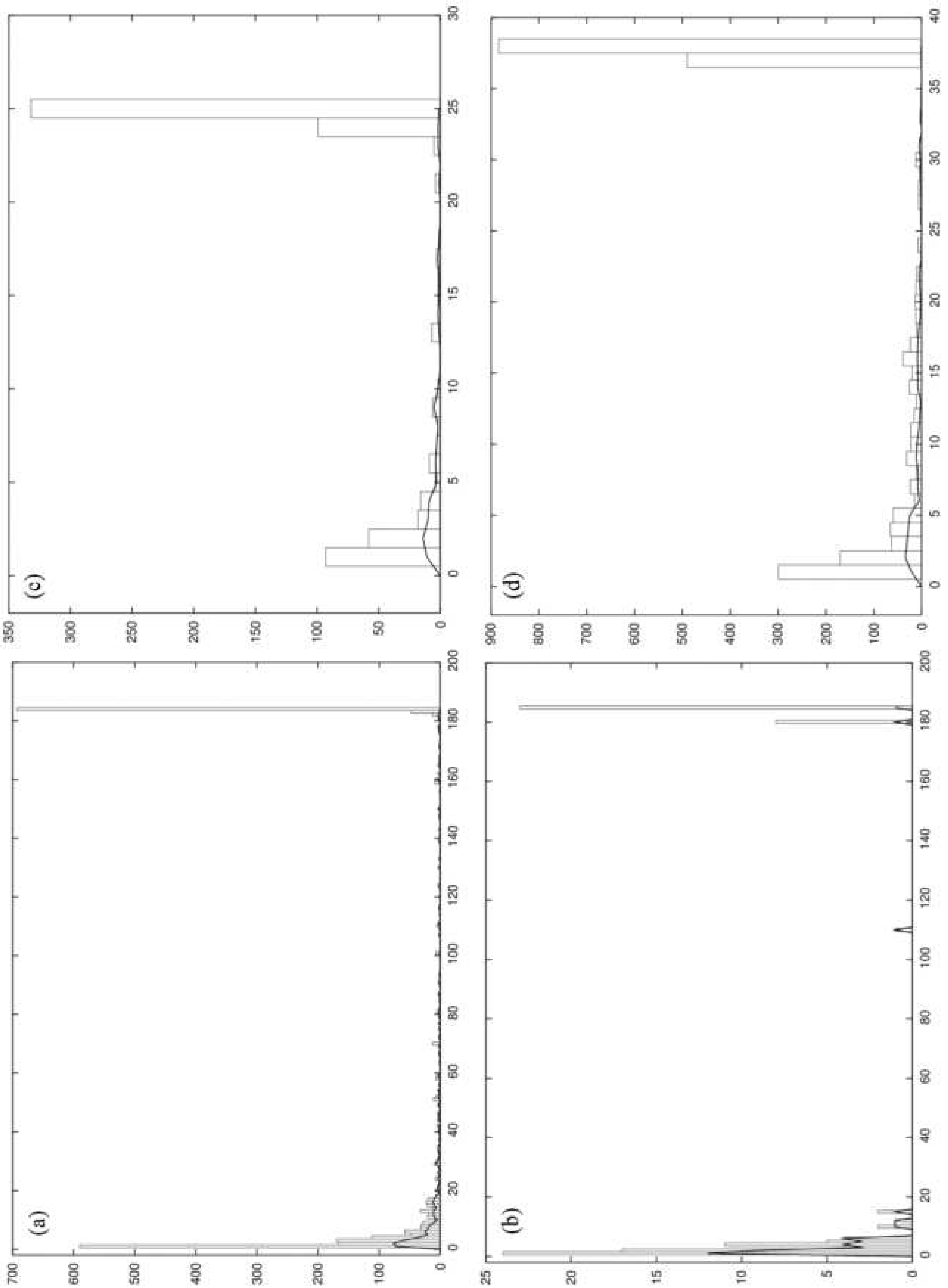


Figure 2. Feature signature histograms for large changes to Python and MzScheme

These are histograms of the number of difference blocks that were executed by exactly each number of different test suites. The line graph shows the number of naïve clusters present among changes with each number of 1's in their feature signatures.

We created these histograms for several other Python changes of similar magnitude. In all these cases, we found graphs that were *nearly identical in shape* to those above, despite also being for semantically unrelated changes.

Figure 2 presents much more significant changes to Python—no longer at the level of a small number of commits, but at the level of significant releases:

(a) The total number of difference blocks is 2442. The change represents the edits between Python version 1.5.2 and 2.2.

(b) The total number of difference blocks is 103. The change represents the edits between Python version 2.2 and 2.2.1.

These changes are different from the previous set in two notable ways:

- The number of difference blocks is much greater, as much as by two orders of magnitude.
- The changes consist of aggregates of from fifty to several hundred commits.

Remarkably, even in these two cases the graphs have the same shape—change blocks concentrated at the left and right ends.

The changes we have seen so far are all edits to the Python source, which leaves open the possibility that these are artifacts of the Python development methodology. We therefore repeated these experiments on an unrelated source code base, namely MzScheme. Figure 2 presents the following changes:

(c) The total number of difference blocks is 668. The change captures the edits between MzScheme versions 102 and 103. The edits took place over two months and mostly consisted of small bug fixes. Thus, while large in scope, it was small in impact.

(d) The total number of difference blocks is 2400. The change reflects the edits between versions 103 and 200, which took 21 months. These are actually consecutive versions; the new version number series reflects substantially revisions not only in the implementation but also to the language itself. Changes included a different object system, a different macro system, a new module system, another garbage collector, and so on. Thus, this was a change of enormous scope.

We see again the same concentration of change blocks at the left and right ends of the histogram. Because many of the changes were to the core language implementation, it is not surprising that the graphs are skewed slightly more to the right than those for Python, for the reason we explain below.

We have also 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 change'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 really do appear to pertain to a very small number of features (i.e., their position is not merely a test suite artifact). We label these *featuristic* changes.

The shape of the typical histogram says that almost all change blocks are either featuristic or infrastructural.

We find it quite remarkable that these and other experiments exhibit such strong similarity. Obviously we cannot extrapolate wildly from these findings, but we find these similarities promising, and suggest that an analysis based on them may apply broadly.

6 Clustering

While feature signatures and their frequencies 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 related ones. Clusters must reflect *conceptual changes*: aggregates of changed blocks that the programmer thinks of as one change to the program. It is of course impossible for a tool to determine these relationships automatically; the best we can do is abduce intent based on similarity, as do many other AI tools. (Fortunately, we envision that the results of clustering will be used primarily to guide humans, not in a purely automatic manner.) 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 a commit a single cluster).

6.1 Naïve Clustering

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 performed an analysis of the naïve clusters for the seven small changes to Python. For these changes, we had good comments by the authors in the change logs about the conceptual changes made; there was always more than one such conceptual change between the versions we studied.

Our study of the naïve clusters suggests that they match well with conceptual changes. That is, in the changes we investigated, all the code blocks grouped into a naïve cluster were clearly identifiable as one 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 conceptually.

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. Since the number of clusters is not very meaningful for small changes, we present them only for the major changes to Python and MzScheme (see the line graphs in Figure 2). (This presentation makes sense because change signatures must have the same number of ones in order to have a chance of being equal, so naïve clusters can only be formed among change blocks that fall in the same bar of this graph.) Reading the graphs left-to-right, the number of clusters begins small, then dwindles to one or zero.

6.2 Other Potential Clustering Methods

While the naïve clusters are a significant improvement over looking at blocks individually, they are "incomplete", in the sense that not all change blocks that are part of the same conceptual edit always fall within the same cluster. We therefore undertook a more careful study of the data. We took several Python program changes and, for each one, manually partitioned the naïve clusters into a small number of conceptual changes. We found that naïve clusters with feature signatures that had a low Hamming distance (the count of placewise discrepancies between the two vectors) tended to group together. However, another prevalent pattern we observed was that naïve clusters such as these two,

	ts 1	**ts 2**	**ts 3**	...	**ts** m
Block 1	1	1	0	...all 0's ...	0
Block 2	1	1	0	...all 1' ...	1

where all test suites that ran Block 1 (the block run by fewer tests) also ran Block 2 (the one run by more tests), often needed to be grouped into the same conceptual edit. These naïve clusters have a high Hamming distance; that is, they do not have a high absolute number of ones in common components. However, two vectors cannot share more ones than the number of ones in the vector with fewer of them, and the percentage of ones that they share relative to this bound is high.

We can exploit this pattern by using a clustering algorithm that is parametrized by a similarity metric and supplying an appropriate metric. We employed the following one:

1. Treat the two feature signatures as vectors of zeroes and ones, u and v of length n, where n is the number of test suites run.

2. Let $count(v) = \sum_i^n v_i$; that is, the number of ones in v.

3. Let $sharedones(u, v) = u \cdot v$. Note that because u and v are vectors of zeroes and ones, this dot product counts the number of ones they share in the same component.

4. The similarity between two vectors is given by

   $$d(u,v) = \frac{sharedones(u,v)}{\rho * min(count(u), count(v)) + (1-\rho) * max(count(u), count(v))}$$

   where a useful value of ρ was 0.8. The higher ρ is, the more important it is that vectors share all possible ones; smaller values of ρ place more emphasis on the absolute number of ones in common.

5. Subtracting the similarity from 1 gives the distance.

We used the similarity metric based on d with the k-mediods clustering algorithm [22], which is parametrized by the similarity metric used to compare vectors, as well as the number of clusters generated (k).

While the results of using d were encouraging, they did not correspond exactly to our manually-constructed clusters. In part, this is because there are several meaningful clusterings of the changes, so disagreement with our manual clustering was not always indicative of a problem. Only in very few cases, moreover, did we consider the automatically constructed clusters wrong. However, one shortcoming of this specific method is that it requires the programmer to propose a number of clusters (the k in k-mediods), though this is a concern common to many data mining approaches.

Consequently, the experimental work in the rest of this paper relies on naïve clusters—underapproximation is generally safer than overshooting, and hand-examination revealed that the naïve clusters were useful underapproximations. However, the d metric shows how knowledge about the structure of feature signatures can be applied in the design of a clustering technique. We have included this presentation of an alternative clustering technique to highlight that advances in clustering techniques can improve our results, and to present a direction for further exploration. 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 Rationale Construction

There are two ways to apply the information from the test suite vectors to the rationale problem. The first is a tool that helps programmers write better rationales when they perform a commit; the second is a related tool that helps later programmers discover rationales ex post facto for poorly-annotated changes.

At its simplest, the feature signature is useful for creating templates of change logs that are kept in version control software. A tool can generate a template for each change block as follows:

```
Objects/classobject.c_4
     [lines 1235-1235 in the new version]
is associated with the features:
  test_class
  test_coercion
```

(The _4 identifies a unique block amongst the changes made to the file. The subsequent lines list the features in its feature signature.)

Given a cluster of blocks that are part of a conceptual edit, we can supply the union of their non-infrastructural feature signatures as the rationale template. For example:

```
Blocks:
 Objects/listobject.c_9 [1364-1364 (new)]
 Objects/listobject.c_8 [1361-1362 (new)]
 Objects/listobject.c_7 [1344-1344 (new)]
 Objects/listobject.c_6 [1341-1342 (new)]
are associated with the features:
  test_types
  test_userlist
```

Generating these templates has two main benefits: (a) it prompts programmers to provide meaningful descriptions of changes, and (b) it reminds programmers of changes they may otherwise forget to document by giving much more fine-grained information than merely which files changed.[2] The templates also help a subsequent code browser identify incomplete logs. Furthermore, by being lightweight and automatic, this process is easy to integrate with a tool such as CVS.

This process is equally useful to programmers attempting to understand a program. Given an unclear code fragment, the programmer gets the changes that impacted the code, and then applies our methodology to pairs of versions; the features that the code impacted at each change then tell the programmer something about how the code evolved to its current state.

[2]In their personal experience as developers, the authors have certainly had instances when they have forgotten about changes made between commits and hence failed to document some of them.

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 [9], Ganter and Wille [18] defined the notion of *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). Concept analysis has been used in other software engineering projects for program refactoring such as that of Siff and Reps [36] and Tip and Snelting [38].

Most changes should carry enough information in the matrix of 0's and 1's to construct concept lattices. We show in Figure 3 the concept lattice that ensues from adding deep object comparison to Python.[3] Studying the concept lattice helps the user understand potentially subtle relationships between the test suites. For instance, the shaded portion of the figure highlights a pair of test suites, `test_format` and `test_unicode` (labeled (a) and (c) in the figure, respectively), with the relationship that whenever the first exercised a difference block, so did the second.

We have manually examined some of the concept lattices to find that the relationships contained in the lattices are in fact semantically meaningful. Instances when they are not point developers to problems with the code, test suite structure or both. In particular, they suggest instances when suites contain tests that are not especially pertinent to the feature that the test purports to capture.

We believe we can 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. Knowing the relationship between test suites helps us determine when we should perform *feature subsumption*. For instance, in Figure 3, the concept labeled (b), `test_sre`, dominates (c), `test_format`; indeed, we found that the `test_sre` column was more important than the `test_format` column when manually clustering.

[3]We thank Christian Lindig for sharing his software for generating concept lattices.

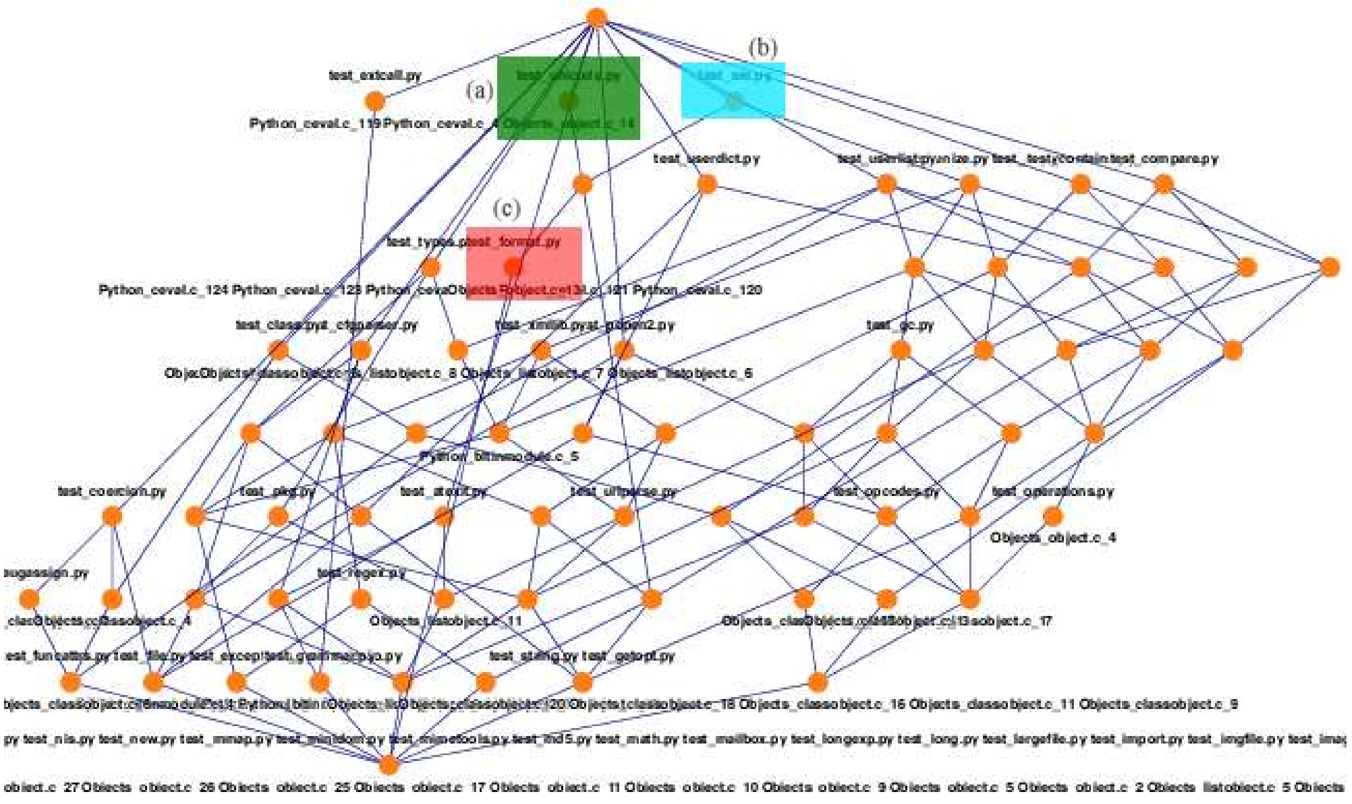


Figure 3. Concept lattice for a Python change

7.3 Visualizing Related Changes

Many code editors and browsers use colors to highlight syntactic "parts of speech". While this may be useful to some programmers, it provides only minimal information about the program's *semantics*; in particular, syntactic coloring merely restates what is fairly obvious from the program source. Visually identifying features is likely to be much more useful to a user. The relatively small number of clusters means the user will not be swamped by an overwhelming panoply of colors.

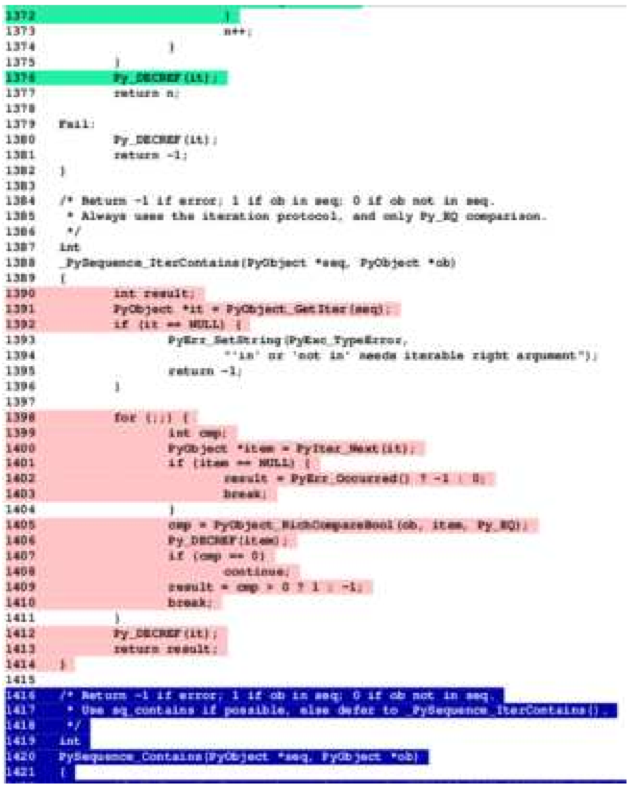
Figure 4 shows the prototype of such a tool operating on a small segment of the Python source. Each *cluster* is given a different color. All lines of code that are given the same color are part of the same feature; syntactically distant but semantically related edits have the same color. A tool like Codesurfer [5] has built-in support for performing such coloring. Natural extensions to such a tool include a "mouse-over" action that displays the feature set corresponding to a particular color.

7.4 Aspect Mining

When developers commit significant changes to a codebase, they often add important new features to the system. We see this in our case studies just from the growth of the number of feature-oriented test suites. Yet each new feature often corresponds to edits to multiple portions of the program source that may not be immediately related. The very fact that multiple difference blocks for the same feature fall within a single cluster immediately suggests this kind of distribution of a feature's implementation, since by definition distinct difference blocks correspond to non-contiguous changes to the code. This corresponds exactly to (and validates) the intuition behind aspect-oriented programming [25].

Our visualization helps programmers to identify these kinds of changes to the codebase. In principle, a programmer should then be able to extract these changes and isolate them into a module.

Unfortunately, this process runs into shortcomings in modern aspect technology. Some forms, such as AspectJ [24], 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 [37] or HyperJ [19], 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



```
                        }
                        n++;
                }
        }
        Py_DECREF(it);
        return n;

Fail:
        Py_DECREF(it);
        return -1;
}

/* Return -1 if error; 1 if ob in seq; 0 if ob not in seq.
 * Always uses the iteration protocol, and only Py_EQ comparison.
 */
int
_PySequence_IterContains(PyObject *seq, PyObject *ob)
{
        int result;
        PyObject *it = PyObject_GetIter(seq);
        if (it == NULL) {
                PyErr_SetString(PyExc_TypeError,
                        "'in' or 'not in' needs iterable right argument");
                return -1;
        }

        for (;;) {
                int cmp;
                PyObject *item = PyIter_Next(it);
                if (item == NULL) {
                        result = PyErr_Occurred() ? -1 : 0;
                        break;
                }
                cmp = PyObject_RichCompareBool(ob, item, Py_EQ);
                Py_DECREF(item);
                if (cmp == 0)
                        continue;
                result = cmp > 0 ? 1 : -1;
                break;
        }
        Py_DECREF(it);
        return result;
}

/* Return -1 if error; 1 if ob in seq; 0 if ob not in seq.
 * Use sq_contains if possible, else defer to _PySequence_IterContains().
 */
int
PySequence_Contains(PyObject *seq, PyObject *ob)
{
```

Figure 4. Code colored according to naïve clusters

occurring in practice. Due to the lack of such technologies, we are unable to provide tool support for this form of aspect mining.

8 Related Work

Our work on code rationales was significantly influenced by past work on design rationales in other engineering contexts. Many other researchers have studied the need for recording design rationales and techniques for doing so. First, various formal languages and processes have been proposed for manually encoding rationales for decisions as they are made. These include *Problem-Centric Design Rationale* [28], *Issue-Based Information System* [12], *Design Representation Language* [27], and *Questions*, *Options*, *Criteria*[29]. While these efforts help structure the rationale discovery process, and can hence lead to substantial documentation, the significant human element means they provide very limited opportunity for automation.

Other techniques provide near-complete automated rationale capture by requiring engineers to adopt new practices and tools. Myers, et al. [33] describe a tool for rationale capture of circuits by observing the designers actions in a CAD tool. Baxter [8] uses program transformations over a clean initial specification to capture rationales. Unfortunately, such tools are deeply intrusive, and are certainly unlikely to work at the Internet scale.

Some design rationale techniques strike a balance between these extremes by requiring some effort on the part of the user but also automating what rationale capture they can. Bahler and Bowen [6] describe a programming language for constructing constraint-based design advisors; these advising programs record rationale in terms of either the constraints satisfied by an action or the constraints that a user is disabling in order to complete an action. Egyed [13] 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 other artifacts. However, the programmer must seed the system with some hypothetical relationships. Program understanding tools such as the Programmer's Apprentice [34] allow the user to tile the program with common code patterns, and thus explain pieces of code in terms of patterns. Murphy and colleagues [32] provide a method for finding where an abstract model of a system and the source code diverge, thus allowing the programmer to check his guesses about the overall program structure. These are extremely different in flavor than the tools we describe in this paper.

Many aspects of our work rely on techniques that have been thoroughly treated in the literature. Clustering is one such technique; Fasulo [15] provides an overview of current clustering techniques, while Kaufman and Rousseeuw [22] describe the k-mediods algorithm that we used. Concept analysis was pioneered by Ganter and Wille [18], and has since been applied to program comprehension and refactoring [36, 38].

In this paper we use `diff` for program comparisons. Yang [41] describes a better program comparison utility. Even better results should ensue by accounting for the program's behavior, but tools that do this [20, 21] tend to be very sensitive to the quality of pointer analyses. While there are some efficient points-to analyses for C, they tend to perform poorly in the face of complex pointer manipulations, which manifest repeatedly in the kinds of applications we have studied in this paper. (In general, there is a tension between the underapproximation resulting from testing and the overapproximation inherent in static analyses.)

Our work is closely related to dynamic slicing [26] and similar tracing techniques. This has mainly been employed for program comprehension and debugging [26] and visualizations [7, 30]. 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 [31] describe a process for refactoring legacy systems into components. 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. They then propose separating code into components by feature by computing the ratio of coverage of each function by each test. Their distinction between "core" and "library" code is similar to ours between featuristic and infrastructural changes. Their experimental work, however, does not consider the evolution of systems over time, nor does it identify evolutionary patterns akin to those presented by our histograms.

Wilde and colleagues [39, 40] describe a dynamic analysis based on test cases for discovering the code implementing a single specific feature. They propose identifying features through tests using two small test suites, one related to the feature and another unrelated, then finding basic blocks exercised by the first suite and not by the second. Their work is, however, limited to a single feature at a time, and does not provide the analysis of feature signatures that we do. Eisenbarth, Koschke and Simon [14] expand this work to multiple features using concept analysis. Though the profiling data they collect and the concept lattice they build are similar to ours, they utilize the concept lattice to guide manual feature discovery based on the program dependency graph rather than to investigate the structure of the test battery.

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] A. Aho, R. Sethi, and J. Ullman. *Compilers - Principles, Techniques, and Tools*. Addison-Wesley, 1986.
[5] P. Anderson and T. Teitelbaum. Software inspection using CodeSurfer. In *Proceedings of the 1st Workshop on Inspection in Software Engineering*, 2001.
[6] D. Bahler and J. Bowen. Design rationale management in concurrent engineering. In *10th National Conference on Artifi cial Intelligence 1992 Workshop on Design Rationale Capture and Use*, San Jose, 1992.
[7] T. Ball. Software visualization in the large. *IEEE Computer*, 29(4):33–43, April 1996.
[8] I. Baxter. Design maintenance systems. *Communications of the ACM*, Vol. 4, April 1992.
[9] G. Birkhoff. Lattice theory. *American Mathematical Society Colloquium Publications*, 25, 1967.
[10] H.-J. Böhm and M. Weiser. Garbage collection in an uncooperative environment. *Software–Practice and Experience*, 18(9):807–820, 1988.
[11] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.
[12] E. J. Conklin and K. B. Yakemovic. A process-oriented approach to design rationale. *Human-Computer Interaction: Special Issue on Design Rationale*, 6(3/4), 1991.

[13] A. Egyed. A scenario-driven approach to trace dependency analysis. *Transactions on Software Engineering*, 29(2), 2003.

[14] 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.

[15] D. Fasulo. An analysis of recent work on clustering algorithms. Technical Report 01-03-02, University Of Washington, 1999.

[16] 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.

[17] M. Flatt. PLT MzScheme: Language manual. Technical Report TR97-280, Rice University, 1997.

[18] B. Ganter and R. Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer Verlag, 1999.

[19] W. Harrison and H. Ossher. Subject-oriented programming: a critique of pure objects. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications*, pages 411–428, 1993.

[20] S. Horwitz. Identifying the semantic and textual differences between two versions of a program. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, volume 25, pages 234–245, White Plains, NY, June 1990.

[21] D. Jackson and D. Ladd. Semantic diff: A tool for summarizing the effects of modifications. In *Proceedings of International Conference on Software Maintenance*, pages 243–252, 1994.

[22] L. Kaufman and P. J. Rousseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis*. John Wiley and Sons, Inc., 1990.

[23] R. Kelsey, W. Clinger, and J. Rees. Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9), Oct. 1998.

[24] 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.

[25] 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.

[26] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, 1998.

[27] J. Lee. Extending the Potts and Bruns model for recording design rationale. In *Proceedings of the 13th International Conference on Software Engineering*, pages 114–125. IEEE Computer Society Press, 1991.

[28] C. Lewis, J. Rieman, and B. Bell. Problem-centered design for expressiveness and facility in a graphical programming system. *Human-Computer Interaction: Special Issue on Design Rationale*, 6(3/4), 1991.

[29] A. MacLean, R. Young, V. Bellotti, and T. Moran. Questions, options, and criteria: Elements of a design rationale for user interfaces. *Human-Computer Interaction: Special Issue on Design Rationale*, 6(3/4), 1991.

[30] A. Malony, D. Hammerslag, and D. Jabalonski. Traceview: A trace visualization tool. *IEEE Software*, pages 19–28, September 1991.

[31] 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.

[32] 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.

[33] K. L. Myers, N. B. Zumel, and P. Garcia. Acquiring design rationale automatically. *AI EDAM*, 14(2):115–135, 2000.

[34] C. Rich and R. C. Wales. *The Programmer's Apprentice*. ACM Press, 1990.

[35] D. J. Richardson, T. O. O'Malley, and C. Tittle. Approaches to specification-based testing. In *Symposium on Testing, Analysis, and Verification*, pages 86–96, 1989.

[36] M. Siff and T. Reps. Identifying modules via concept analysis. In *International Conference on Software Maintenance*, pages 170–179. IEEE Computer Society Press, 1997.

[37] Y. Smaragdakis and D. Batory. Implementing layered designs and mixin layers. In *European Conference on Object-Oriented Programming*, pages 550–570, July 1998.

[38] G. Snelting and F. Tip. Understanding class hierarchies using concept analysis. *ACM Transactions on Programming Languages and Systems*, 22:540–582, May 2000.

[39] 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.

[40] N. Wilde and M. Scully. Software reconnaissance: Mapping program features to code. *Software Maintenance: Research and Practice*, 7:49–62, 1995.

[41] W. Yang. Identifying syntactic differences between two programs. *Software–Practice and Experience*, 21(7):739–755, July 1991.