Dynamic Feature Traces: Finding Features in Unfamiliar code

Andrew David Eisenberg
Department of Computer Science
University of British Columbia
Vancouver, BC, Canada
ade@cs.ubc.ca

Kris De Volder
Department of Computer Science
University of British Columbia
Vancouver, BC, Canada
kdvolder@cs.ubc.ca

Abstract

This paper introduces an automated technique for feature location: helping developers map features to relevant source code. Like several other automated feature location techniques, ours is based on execution-trace analysis.

We hypothesize that these prior techniques, which rely on making binary judgments about a code element's relevance to a feature, are overly sensitive to the quality of the input. The main contribution of this paper is to provide a more robust alternative, whose most distinguishing characteristic is that it employs ranking heuristics to determine a code element's relevance to a feature. We believe that our technique is less sensitive with respect to the quality of the input and we claim that it is more effective when used by developers unfamiliar with the target system.

We validate our claim by applying our technique to three open sourced software systems and five actual tasks, comparing these results to the results of an existing technique. Each of these systems has an extensive test suite. A developer unfamiliar with the system spent a limited amount of effort to prepare the test suite for analysis. The experiment demonstrates that under these circumstances our technique provides more useful information to the developer.

1. Introduction

During software maintenance, developers are faced with tasks such as bug fixes and feature enhancements to meet evolving requirements. The description of these tasks are typically generated from the user base of the system [16] and are therefore most often expressed in terms of *features*. In this context features are defined as behaviors that are observable to users at their particular level of interaction with the system. Developers, however, need to implement the requested user-level feature modifications at the source code level. Thus, mapping between features observable to a user and the code implementing those features is a recurring issue in Software Engineering and is termed *feature location* [14].

There are previous techniques that can assist developers with the onerous task of feature location. These techniques can roughly be classified into *interactive* [4, 5, 11, 12] and *automated* [8, 14, 15, 17]. Interactive techniques rely on a feedback loop between developer and tool, where a developer discovers more information about the implementation of a feature through interacting with the tool. Automated tools, on the other hand, perform a one-time analysis after being provided an initial input. This paper makes contributions specifically in the area of automated feature location. Therefore, we focus on such systems alone.

Automated feature location techniques can further be divided into those that employ only static analysis [17] versus those that employ primarily dynamic analysis [8, 14, 15]. Since we defined a feature as a behavior of a system, we employ dynamic analysis for ours.

A key goal in the design of our technique is to ensure that it is useable by developers unfamiliar with the target system. To be practical in this setting, the technique should be feasible when used with pre-existing test suites such as those that are typically available for systems developed with a Test Driven Development (TDD) strategy [1, 2].

We now examine existing techniques and how they fall short in meeting this key goal. In particular, we examine Software Reconnaissance [13, 14] and Execution Slices [15]. Both are automated feature location techniques that use dynamic analysis of test suite execution. Both depend on the availability of two sets of test cases (two *test sets*): one that exhibits the feature (the exhibiting test set) and one that does not (the non-exhibiting test set). The two test sets are executed and their footprints—the set of code elements executed by a test set—are collected. The techniques then make binary decisions about which elements are deemed relevant. In this context, a binary decision is one that can be formulated as a boolean expression in terms of set inclusion of a test set footprint. Typically, they subtract the non-exhibiting footprints from the exhibiting foot-

prints. The remaining code elements are the set of *U*nique *COMP*onents, or *UCOMPS*,[1] that uniquely implement a feature by being exercised in at least one of the exhibiting tests and none of the non-exhibiting tests. Intuitively, the quality of the result depends on the degree of correlation that exists between exhibiting and non-exciting tests. The ideal non-exhibiting test set is correlated to the exciting test set in the following way:

1. It exercises at least that part of the exhibiting footprint that is *not* directly related to the feature.

2. It does not in any way exercise the code directly related to the feature.

Test suites developed under TDD are a good source of exhibiting tests. However, they are *not* a good source of correlated non-exhibiting tests—indeed the TDD methodology does not call for the explicit construction of "tests that do not test feature X". Because of this, we believe that techniques like Software Reconnaissance and Execution Slices cannot be used effectively with a typical test-suite developed under TDD, unless developers are willing and knowledgeable enough to develop correlated non-exhibiting tests themselves.

The main contribution of this paper is to provide an alternative automated feature location technique, which we call *Dynamic Feature Tracing*. Like the prior techniques it uses execution-trace analysis. The most distinguishing characteristic of our technique is that it employs gradual ranking heuristics to determine a code element's relevance to a feature as opposed to binary judgments.

We claim that a Dynamic Feature Trace (DFT) is less sensitive to the degree of correlation between exhibiting and non-exhibiting tests and therefore produces more useful results than previous techniques when using an existing test suite as the source of exhibiting as well as non-exhibiting tests.

We validate our claim by applying DFTs to test sets that are produced by a developer unfamiliar with the target system who spends a limited amount of time partitioning an existing test suite. The results of our experiment show that under these assumptions DFTs provide more useful information to the developer than a technique based on binary judgments.

The remainder of this paper is structured as follows. In the next section we introduce DFTs and explain how they are created. In Section 3, we describe how DFTs can be visualized and explored. The main experiment validating our claim is described in Section 4. In Section 5 we describe a case study, providing additional anecdotal information about how our technique may be used in practice. Section 6 discusses some additional details about related work. Section 7 summarizes the paper.

[1]This is the terminology used by Software Reconnaissance but the Execution Slices technique uses a similar formula.

2. Dynamic Feature Traces

We have created a prototype DFT tool to create DFTs from a JUnit [9] test suite of a Java[2] system, given a mapping of test cases to test sets and then to features. This mapping must be defined by the developer, but is extracted from a test suite that is correlated to a feature set, which is typically present in those systems developed using TDD. To the extent that this correlation exists, our technique can be used.

2.1. What is a DFT?

A DFT is comprised of two parts: 1) the *ranks*, an ordered set of all methods called during the execution of the test set footprint and 2) the *calls*, a set of all observed method calls amongst the methods in the test set footprint.

The *calls* set retains some information about the calling context of particular elements within a test execution. We believe that making this information available to the developer may help ascertain why a method is relevant to a feature.

DFTs are created in three steps. First the developer partitions the test suite (Section 2.2). Subsequently, the DFT tool performs execution trace analysis (Section 2.3) and then creates the ranks and the calls sets (Section 2.4). DFT creation is automated after the initial feature-test mapping has been specified.

2.2. Partitioning the Test Suite

The DFT creation process expects a system with a large and fairly comprehensive test suite that has a correlation between features and test cases (i.e., all features are tested in at least one test case).

The developer provides the DFT tool with a feature mapping that as much as possible—given that the developer has imperfect knowledge of the system—groups all test cases that collectively and comprehensively exhibit all parts of a feature; this is the exhibiting test set. Creating this exhibiting test set can be accomplished by, for example, browsing the test suite and searching for words in the code that match or describe the feature. These mapped features are the ones explicitly analyzed by the tool.

All remaining tests not explicitly in the mapping are implicitly grouped into test sets based on similarities in the test suite structure—the test classes themselves. This implicit mapping is used to compare the exhibiting test sets to the rest of the test suite. The developer can also explicitly

[2]Java is a trademark of Sun Microsystems

exclude test cases or test classes from the analysis. Hence, for each feature, there is one exhibiting test set and many non-exhibiting.

The artifact created at this stage is a feature mapping file which defines a mapping from features to test cases. For convenience, the DFT tool can analyze multiple features in a single pass if they do not share test cases. Otherwise, however, the DFT tool must be run separately for each feature.

2.3. Gathering the Execution Trace

The DFT tool performs the profiling and analysis of the tests and is written in AspectJ [7], an extension of the Java programming language that provides aspect-oriented programming facilities.

The DFT tool first collects the execution trace; then it generates the *calls* and *ranks* sets (described in the next section). During each test execution, unless a test is explicitly excluded from the trace, all method calls and their call depth are stored by the tool.

The artifact created from this stage of the process is a data structure that contains all information needed to calculate the *calls* and *ranks* sets: a record of all method calls, the test case and test set they are parts of, and the depth of those calls.

2.4. Calls and Ranks

Using the data gathered by the execution trace, the DFT tool generates the *calls* and *ranks* sets, created for each feature analyzed.

After the *calls* and *ranks* are created, the DFT is stored in an intermediate format that can be the input to exploration and visualization tools, which is described in Section 3.

2.4.1 Generating the Calls

For each test set explicitly defined in the mapping, one *calls* set is created, based on the execution trace gathered by the tool. Each element of the *calls* set is a tuple, containing the caller method signature, and the callee method signature. Methods are added only once for each caller/callee pair in a particular test set, even if the same call occurs multiple times.

2.4.2 Generating the Ranks

Each element of the *ranks* set is another tuple containing a method signature, and its rank in the feature. There is one entry for each method executed in the test set. In order to determine the rank of each method in a test set, the method is given a score from 0 to 1—the average of three heuristics each scoring the method from 0 to 1. The final rank of a method is determined by sorting each exercised method by this score, with the top scoring method given a rank of 1, the next a rank of 2 and so forth.

We developed the three heuristics iteratively using the source code of a TDD system as a base. We posited a heuristic, created a DFT for a feature using only that heuristic, and compared the *ranks* set to what we had determined to be relevant methods. Through this process, we were able to try different variations of heuristics, iteratively refining and pruning them until we were satisfied that they successfully captured sufficient information about the feature.

For each heuristic, we first describe the intuition behind it and then we describe its implementation:

Multiplicity: A method exercised by a test set multiple times and in different situations is more likely to be important to the exhibited feature than a method used less often. We therefore compute *Multiplicity*: the percentage of exhibiting tests that exercise a method compared to the percentage of methods in each non-exhibiting set of tests will contribute to its score. In the equation below, the *least* and *greatest* exercising test sets are the test sets that respectively exercise the method the least and greatest number of times.

$$\begin{aligned} Let\ curr\% &= \textit{\% of tests that exercises method in} \\ &\quad \textit{the current test set} \\ min\% &= \textit{\% of tests that exercises method in} \\ &\quad \textit{the least exercising test set (can be 0)} \\ max\% &= \textit{\% of tests that exercises method in} \\ &\quad \textit{the most exercising test set} \end{aligned}$$

$$Multiplicity = \frac{curr\% - min\%}{max\% - min\%} \quad (1)$$

Specialization: A method that is exercised by many different features' test sets is more likely to be a utility method and therefore not a part of any feature at all, than a method that is exercised by fewer test sets. Thus, we compute *Specialization*:

$$Specialization = (1 - \frac{\textit{\# of exercising test sets} - 1}{\textit{\# of number test sets}}) \quad (2)$$

Note that the -1 is added to ensure that methods exercised by only a single test set has a score of 1.

Depth: We expect that for a well-designed and well-partitioned test suite, a test set will exhibit the behavior of the feature it focuses on in the most direct manner. This is not to say that other features are not more directly exhibited by the test set (perhaps by being a prerequisite for the

feature focused upon), but rather, that all other test sets exhibit the feature no more directly than this one. We correlate directness with call depth.

This heuristic is partially implemented by comparing minimum call depths of an exercised method in all exercising test sets. The rationale is that the more directly a method is exercised, the shallower its call depth.

If the *Depth* heuristic were to examine only the test sets that exercise the method (ignoring the non-exercising tests), the resulting score would be sensitive to this number of exercising tests—an undesirable effect. This would mean that methods exercised by only a handful of test sets could not be meaningfully compared to those methods exercised by almost all. For this reason, the *Depth* calculation examines *all* test sets, even those that do not exercise the method.

To find the $Depth$ score for a method, m, called with a stack depth of $calldepth$ in a test set:

1. Determine the lowest and highest (low, hi) call depths (fewest or greatest number of stack frames from the top) for m in all test sets that exercise it.

2. Determine the linear scale, $scaled$, of $calldepth$ between low and hi: $scaled = 1 - \frac{calldepth-low}{hi-low}$, thus $s = 0$ for the test set with low call depth, and $s = 1$ for the test set with hi call depth. If $low = hi$ (i.e. all test sets have the same call depth), then we define $scaled = 0.5$ (avoiding the division by 0).

3. Next, re-scale $scaled$, taking into account all test sets that do not exercise m, using the $Specialization$ score.

$$Depth = (scaled \cdot (1 - Specialization))+ Specialization \quad (3)$$

Thus, the *Depth* score ranges from the *Specialization* score up to 1. The feature in which method m has *Depth* = *Specialization* exercises m at the *lowest* call depth, while the feature with *Depth* = 1 exercises m at the *highest* call depth. In all other features, m has a *Depth* score linearly scaled between the two.

Having the *Depth* score be dependent on the *Specialization* score is one way to ensure that the *Depth* score is derived from all test sets, even those that do not exercise the method.

We chose these three heuristics based on our intuition of how methods that are relevant to a feature would be executed in test cases that belong to a feature's test set. We show in Section 4 that these heuristics, taken together, are reasonable because they produce DFTs containing valuable information for developers. However, we leave the evaluation and optimization of individual heuristics as future work.

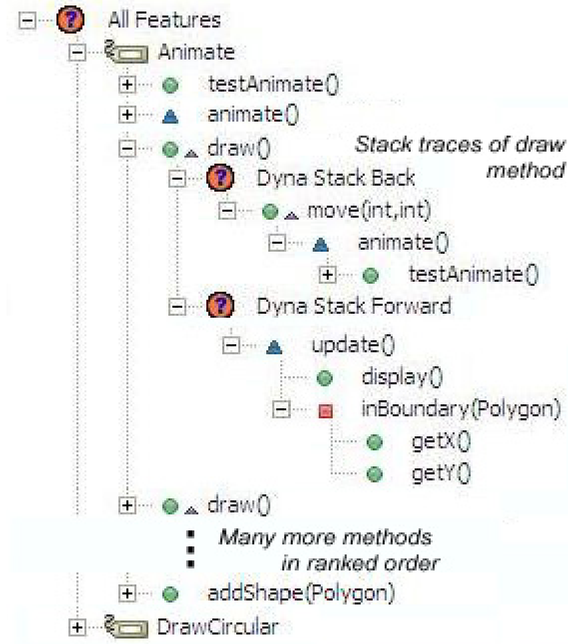

Figure 1. Part of the *All Features* browser, showing the methods of the `Animate` feature in ranked order. The `Animate` feature contains two additional sub-browsers: the *Dynamic stack forward* and *Dynamic stack back* sub-browsers. The fully qualified names can be switched on if desired.

Our claim is not that these are the best possible heuristics to rank the methods, but that given these heuristics, our technique does produce viable results. Future experiments will be needed to fine-tune these heuristics.

3. Viewing and Interacting with DFTs

DFTs can be visualized in JQuery [6],[3] a generic, extensible code browser built as a plug-in for the Eclipse IDE.[4]

A developer can use JQuery to create a browser that displays a set of code elements (fields, methods, classes, etc.) that satisfy an expression based on static relationships between them (calls, reads, writes, inherits, etc.). This is a generic way to describe call graphs, inheritance hierarchies, etc. Each element of a browser can be explored in further detail by creating a sub-browser using that element as a starting point, allowing a developer to incrementally explore the code base. Double-clicking on any code element in the browser opens an editor focused on that piece of code.

We expanded JQuery's functionality so that it could import the *calls* and *ranks* sets of a DFT and create browsers based on them. Figure 1 shows an example of these DFT

[3] http://jquery.cs.ubc.ca/
[4] http://www.eclipse.org

browsers, extracted from a simple drawing application.

The top-level browser is an `All Features` browser, which returns all methods exercised in all DFTs in ranked order. Here, the `Animate` feature is expanded and `DrawCircular` feature is at the bottom, but is collapsed. The information used to create the *All Features* browser in JQuery is drawn directly from the *ranks* set.

The `draw()` method's dynamic call stack is further explored using sub-browsers. The `Dynamic Stack Forward` sub-browser displays the dynamic call graph starting from the selected node and moving forward in the call graph (to prevent infinite recursion, only the first 5 frames are shown). Also shown is a `Dynamic Stack Back` sub-browser. In a similar way, it shows the backwards trace of the dynamic call graph. The information used to create these browsers is gathered directly from the *calls* set.

The three browsers we have defined in JQuery to explore DFTs helps developers quickly see which methods were ranked most highly in the DFT and also allows a developer to further explore any methods that seem to be relevant to a given task.

4. Evaluation

In this section we discuss the study we performed to validate the our claim: that compared to prior techniques based on dynamic analysis of exciting and non-exhibiting test set execution, DFTs are more effective when used by a developer who is unfamiliar with the target system and who expends only a moderate amount of effort to prepare its existing test suite for analysis.

To validate this claim, we compare DFTs to UCOMPs. The computation of UCOMPs is taken to be representative of the techniques like Software Reconnaissance and Execution Slices, that use binary decisions based on the footprints of exhibiting and non-exhibiting test sets. In addition to UCOMPs, Software Reconnaissance and Execution Slices also proposed other binary formulas for locating relevant code. However, they determined that UCOMPs are the most useful for identifying code relevant to a particular feature.

4.1. Assessing Quality of Output

Validation of our claim will require passing judgment on the quality of the output produced by automated feature location tools. Here we formulate the guidelines we used to make this judgment as objectively as possible.

There is no objective way to precisely map features to code in a way that all developers of the system can be in complete agreement on [15]. This complicates making objective judgements on how well code elements suggested by a feature mapping capture a feature's implementation. To avoid this issue, we assess quality of output in terms of its potential utility for a developer performing a specific feature enhancement task, rather than how well it captures all the code implementing the feature.

Change tasks can be precisely characterized by mining a CVS [3] repository of a system and comparing code diffs between successive versions. Hence, we can measure how much of the code that needs to be changed in order to complete the task can be discovered using the feature location techniques.

To judge utility, we model the usefulness of elements suggested by a feature location technique in two ways. First, most interesting and directly relevant are those code elements that are modified in the process implementing the enhancement. We call this set of code elements the *core* set. Second are those code elements directly related to the core set because they call, are called by, override, or are in the same class as the core set. While these elements are not necessarily directly relevant for the task at hand, they may aid in discovering elements in the core set. We call the union of the core set with this set of related elements the *expanded* set.

To judge the usefulness of results suggested by a feature mapping technique, we will consider the likelihood that it can bring elements in either the core or expanded sets to the attention of the developer. **A technique is considered most successful if it brings elements in the core set to the developer's attention**. If a technique is unable to suggest elements from the core set, we examine the expanded set. We consider the technique only moderately successful if it may bring elements from this larger set to the attention of the developer.

4.2. Systems Analyzed

We chose three open source Java systems to analyze created by different developers:[5]

HTMLUnit a unit testing framework focusing on testing dynamically generated web pages; about 17 KLOCs with 2,410 methods and constructors. The test suite has 86% branch coverage and 66% line coverage.[6]

HTTPUnit a unit testing framework that models the HTTP protocol; about 16.5 KLOCs with 2,227 methods and constructors. The test suite has 88% branch coverage and 80% line coverage for the test suite.[7]

[5] The statistics quoted correspond to the systems' states immediately before the latest feature enhancement we analyzed was checked in.

[6] http://sourceforge.net/projects/htmlunit

[7] http://sourceforge.net/projects/httpunit

System	Feature	Description
HTMLUnit	RFE 805051	Add support for more operations on *DOM* nodes.
HTTPUnit	RFE 638311	Add support for `document.open()` and `document.close()` to Javascript.
	RFE 717752	Ability to retrieve page links by regular expression.
Axion	Issue 5	Boolean typed columns should be comparable.
	Issue 9	Should be able to have multiple columns of type *CLOB* in a table.

Table 1. Features Analyzed. RFE means "Request for Enhancement".

Axion a file-based database implemented in Java, 43 KLOCs, with 6,216 methods and constructors. The test suite has 90% branch coverage and 76% line coverage.[8]

These three systems were chosen because their source repositories are freely available, they were developed using a Test Driven Development strategy (making their test-suites large and fairly comprehensive) and they all have an active feature request list from which we could relate feature enhancements to CVS check-ins.

From these systems, we searched for completed feature enhancements. Table 1 provides a description of the 5 enhancements we analyzed. The enhancement names we use are the same as those on the feature tracker hosted on the development sites. There was no prior knowledge of the code-base of any of these systems prior to the study. Furthermore, there was little formal documentation provided for developers. All that existed were CVS logs, sparse newsgroup discussions, bugzilla-like feature and bug trackers, and a test suite.

Although the tasks we picked are fairly low-level enhancements to a system, they do constitute feature enhancements in the sense of our earlier definition of feature, since they are requests made by users of the systems and are therefore formulated in terms of "behaviors that are observable to users at their particular level of interaction with the system". The enhancements are also roughly the same granularity as those used in the case study of [14] making them a fair benchmark for comparison.

4.2.1 Data Collection

The process for collecting the data was identical for all features and was performed by an author of this paper:

1. We read the feature request page for the feature which includes information posted by a user about the desired behavior of the enhancement to the system.

[8] http://axion.tigris.org

2. We performed a checkout of the code from the CVS repository corresponding to the time immediately before the feature request was closed.

3. To create the DFTs:

   - We partitioned the test suite into an appropriate exhibiting test set for the feature, using as much relevant information as possible (from the feature tracker, etc.), but without spending time to understand code details.

     In particular, we performed this task by searching the test classes looking for names that matched key words of any relevant information we discovered. From there, we looked through the test methods of those classes and occasionally browsed the code of the methods, making our decisions based on this information.

   - We compiled and wove the code with the DFT tool, and then executed the test suite. The DFT tool collected data based on the results of the execution and applied the ranking heuristics to the data collected during the execution.

4. To perform the binary feature location technique:

   - We started by using the same exhibiting test set used to create the DFTs. To create the non-exhibiting test set, we followed the same test suite browsing process as before—start with test class names, explore test methods, and only rarely explore test code. We examined the remainder of the test suite, from this we removed obviously conflicting tests. All others were put into the non-exhibiting test set.

   - We then re-executed the test suite, collecting the exhibiting and non-exhibiting footprints. We subtracted the non-exhibiting footprint from the exhibiting footprint to retrieve the elements uniquely exercised by the feature, the UCOMPS.

5. Using CVS tools, we generated the *core* and *expanded* sets of methods corresponding to the feature enhancement.

6. We compared the DFTs and the UCOMPS to the *core* and *expanded* sets.

We performed these experiments on a Pentium 4 with 764 Mb RAM and a 2.4 GHz processor running Windows XP. The AspectJ compiler used for this experiment was version 1.1.1. No more than 15 minutes were spent creating the exhibiting test sets and another 5 were spent creating the non-exhibiting test sets for each feature enhancement.

The actual execution time of the test suites increased by about twofold when the DFT tool was woven into the code. HTTPUnit and HTMLUnit's test suites each executed in over 2 minutes without the DFT tool's involvement, and just under 4 minutes each with the involvement. Axion's test suite ran in just under one and a half hour without the DFT tool and about 3 hours with the tool.

4.3. Results

The goal of this experiment is to compare the two techniques under the same conditions using roughly the same amount of effort and the same initial understanding of the system. To validate our claim, the DFTs need to be clearly more useful than the UCOMPs. The results of the experiment are summarized in two tables.

Table 2 shows data about size of the core and expanded sets and their relationship with the exhibiting test set footprint.

Table 3 shows the results of the comparison. For UCOMPs, the table displays both the total number of methods marked as UCOMPs, as well as the number of those methods that are part of the core or expanded set respectively. For DFTs, we expect that a user of DFTs will tend to closely analyze the top results, but skim down the list, ignoring the lowest results. Therefore, DFTs are presented in the "DFT # found in top n" columns. Each column shows the number of relevant methods found in the top 10, 20, 30, 40, and 50 elements respectively. Elements beyond the 50th are not included. Non-integer values occur in the table because we proportionally compute the ranks of equally scoring methods that cross the 10, 20, etc. boundaries.

We used these numbers as the basis for assessing whether or not the results are potentially useful to a developer. Our assessment is noted in the columns labeled "Useful?". We enter one of:

Core signifies that elements in the core set are likely to be discovered by the developer. This is the case we consider as having the *most potential* to be useful to the developer.

Expanded only signifies that a technique fails to pick out any core methods but may bring methods from the expanded set to the developer's attention. This case we consider as having only *moderate potential* to be useful to the developer.

No signifies that a technique fails to pick out any (core or expanded) methods. This case has *no potential* of being useful.

Table 3 shows that DFTs are more useful than UCOMPs. DFTs are successful in finding core methods in 3 out of 5 tasks and find methods in the expanded set only in a 4th task. UCOMPs on the other hand fail to identify a single core method and only find methods in the expanded sets in 3 out of 5 cases. This means that not only do DFTs return useful result in more of the tasks, it also indicates that the results are qualitatively better since they are more directly related to the task at hand.

It is important to note that although the DFT for the core set of RFE 717752 only discovered 1 method in the first 20 to 30 ranks, this is the *only* method of the core set in the exhibiting footprint of 291 methods. For this reason, we marked the core set as useful.

Although we have only performed the comparison on 3 systems, we believe that because these systems are authored by unrelated groups of programmers, they are reasonable representatives of open source TDD systems of their size.

These results demonstrate a noticeable benefit of DFTs over UCOMPs; thus substantiating our claim.

4.4. Further Discussion of DFTs

The results in the previous section show that our DFT technique is noticeably better than the UCOMP technique when applied to unfamiliar systems. However, the results did show that some DFTs missed a significant number of interesting code elements. In this section we make some observations as to why this happened.

RFE 638311 Core This DFT did not exercise 4 methods of the core set. However, none were executed from any test in the entire test suite. Hence, the test suite itself was not complete. They were all variations of constructors that simply performed some initialization and called other constructors.

RFE 717752 Core Of the 5 methods of the core set that were not exercised in the exhibiting test set, all 5 of them were exercised in the JUnit test class `HtmlTablesTest`. Had this test class been included in the study, then all missing methods would have been found.

Issue 9 Core This is an anomalous case; the only feature for which the heuristics appear to be ineffectual. Although the methods were exercised by the exhibiting test set, they were not ranked highly. Trying to explain this anomaly, we discovered that we had missed a test case in the test set for the feature enhancement. This was not an initially obvious test case and had no apparent relationship to the feature. After adding the missing test case to the test set and re-running, the new ranking for Issue 9 Core was improved.

These results are not included in the table because we were specifically interested in assessing how DFTs would

Feature Name	Size of Exhibiting Test Set Footprint	Size of Core / Expanded Sets	Total # Core/Expanded Methods Exercised in Exhibiting Footprint
RFE 805051 Core	215	20	7
Expanded		84	62
RFE 638311 Core	304	5	1
Expanded		61	40
RFE 717752 Core	291	6	1
Expanded		32	17
Issue 5 Core	231	13	13
Expanded		65	62
Issue 9 Core	463	4	4
Expanded		73	64

Table 2. Feature/Test Set Data.

	UCOMP Data			DFT Data					
Feature	# UCOMPs Suggested			DFT Relevant Suggested in top n					
Name	Relevant	Total	Useful?	$n =$ 10	20	30	40	50	Useful?
RFE 805051 Core	0	0	No	2	2	2	4.5	7	Core
Expanded	0	0		3.5	12	14	16	21	
RFE 638311 Core	0	7	Expanded	0	0	0	0	0	Expanded
Expanded	3	7	Only	1.5	3	4	6	10	Only
RFE 717752 Core	0	19	Expanded	0	0.5	1	1	1	Core
Expanded	4	19	Only	0	0.5	3.5	4.5	6.5	
Issue 5 Core	0	12	Expanded	0.33	1.67	5	7	8	Core
Expanded	1	12	Only	3.33	6.66	14	19	22	
Issue 9 Core	0	0	No	0	0	0	0	0	No
Expanded	0	0		0	0.5	1	1.33	1.66	
Useful results for	**0 Core + 3 Expanded / 5**			**3 Core + 1 Expanded / 5**					

Table 3. Comparison of DFT results to the binary technique.

perform when used by unfamiliar developers who are likely to make exactly such mistakes.

For the problematic DFTs described above, the difficulties arose due to either problems with the test suite, or a poor partitioning. Developers unfamiliar with the system have no control over the test suite quality, but can create better DFTs by spending additional time partitioning, as we have shown above. Still, even with our naïve partitioning, many elements of the core and expanded sets were in the test set footprints and ranked highly.

5. Case Study

In this section, we describe a case study we performed that applies a DFT to a feature enhancement on an existing system. We show how DFTs capture information about features not easily available from standard sources.

The task was a feature enhancement on the HTTPUnit codebase, corresponding to a CVS check-in on December 12, 2002. The enhancement consists of adding the ability to *get* and *set* cookies using the Javascript `document.cookie` statement. The task was carried out by an author of this paper who had little knowledge of the code base prior to starting the task. We will summarize the approach taken, how we created and used DFTs, and highlight several cases where DFTs were particularly useful.

Performing The Task: Adding Cookie access to Javascript After reading the relevant newsgroup thread regarding this enhancement, we realized that the enhancement consisted of two parts, corresponding to two features: manipulating cookies, and calling statements on the Javascript `document` object.

With this in mind, we created 2 feature test sets: `Cookies` and `DocumentGetSet`. We spent 10 minutes browsing the test suite to choose two test cases that seemed appropriate for testing general cookie functionality. Later, we spent another 5 minutes to choose 3 test cases for the second test set.

After creating the DFTs and importing them into JQuery, we started by exploring the ranked methods, starting from the highest rank. For both DFTs, we discovered 2 methods in each DFT, all in the top 10 of their DFTs, that seemed directly relevant to the features. These methods became the seeds of further exploration. After finding these initial seeds, we used the dynamic call graphs extensively to determine how these methods were involved with the feature.

To complete the task, we altered or added a total of 8 methods in about 2 hours. Although only a small amount of code needed to be changed, the code was scattered throughout the system, requiring an understanding of several components of HTTPUnit.

Using the DFTs helped highlight information about the implementation of features. The rankings tended to emphasize methods that were relevant to the feature being explored and the calls helped explain how these methods were exercised within the context of the feature. Next, we illustrate four specific instances in this task where we used information contained in DFTs not readily available from

standard search tools.

Textual Relevance Some of the relevant methods that were used to explore the feature were explored simply because of their prominence in the ranking scheme, but were not textually related to any part of the feature description and thus could not have been found using textual searches for words related to Javascript.

For example we used the methods `postAlert()` and `getNextAlert()` as starting points for exploration into Javascript object manipulation. They were explored initially because of their prominence in the ranking of the test set and they were deemed important after some exploration of their dynamic call graphs.

Relevance of Ranking Heuristics Despite their being useful, the overall ranks were not decisive, i.e. not all the top-ranked methods were immediately relevant; conversely, not all relevant methods were top-ranked. Regardless, we were able to complete the task since some relevant methods were identified and ranked highly, providing us with a seed for initial exploration. Also, the dynamic call graph provided enough of a context to help quickly sift through less-related, but highly ranked methods.

Looking for Cookies Had we not created a DFT to help us understand how cookies are implemented, we could have used IDE search functionality instead. However, we would have had to wade through more irrelevant code. For example, a case-insensitive search of the methods named `*cookie*` yields 113 results, with 65 declared within HTTPUnit code and the rest declared in library code. It would have been possible to search through these results, but would have been time consuming and more difficult to determine each method's relevance to the task at hand.

Compare this to the rankings from DFTs, which ranked two relevant methods (`addCookie` and `removeCookie`) in the top ten. From this we were immediately able to narrow our search to those two methods and explore dynamic call graphs from there.

`get`-ting and `set`-ting Properties in `document` Object Using the DFT, we were able to discover that properties and functions in Javascript are *get*, *set*, or *executed* by reflexively calling the entity of that name. For example, a page's links are *get* and *set* by calling the methods `jsGet_link` and `jsSet_link` on the `Document` class.

Whereas, the `Document` class could have been found by using standard search tools, the actual mechanism for reflexively calling Javascript properties could not have been easily understood using only this tactic. This is because the methods in the `Document` class are only called reflexively via a callback mechanism in library functions.

At one point during the task, we had surmised that the method `performEvent` was somehow relevant to the task. Exploring the static call graph in any call explorer from this method would not be effective since the

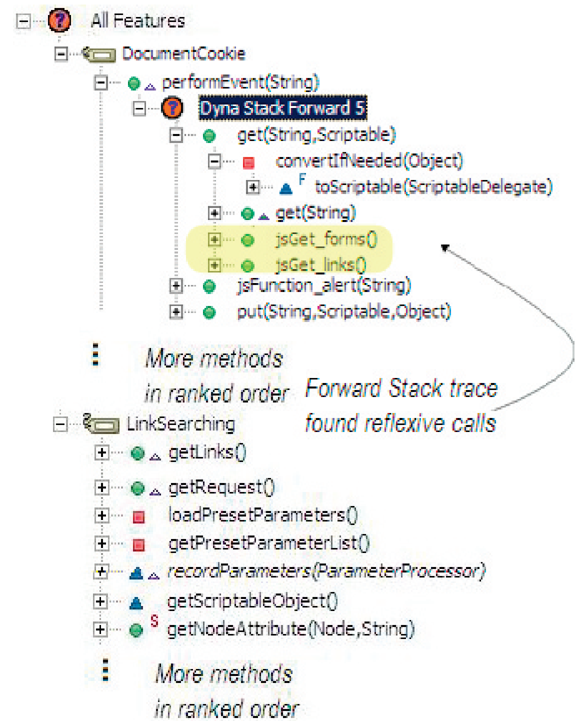

Figure 2. A condensed screenshot of JQuery showing the All Features browser. Some of the ranked methods have been omitted for brevity. Also shown is the dynamic call graph emanating from `performEvent`. Understanding how the two methods pointed to were called helped us understand how Javascript properties are accessed.

`jsSet_xxx` and `jsGet_xxx` methods (what we eventually needed to find) are called only reflexively, not directly. Recall that we were unfamiliar with the system and hence had no knowledge that a callback mechanism was used at all.

Exploring the dynamic call graph stored in the DFT for the same `performEvent` method in Figure 2, the dynamic structure is more apparent and therefore the relationships between `performEvent` and the Javascript manipulation methods can be discovered more easily. Notice that the DFT does not show the library code, but rather just shows the methods as they were executed within the context of the feature.[9]

6. Related Work

In this section we discuss some additional details about related work, complementing what has already been discussed in the introduction.

[9]It is possible to include the library code in the DFT. But, this greatly expands the size of the footprint, and was not the desired behavior for this task.

Licata, et al [8] compare execution traces of entire test suites between versions. The differences between the traces are called *Feature Signatures*. This approach has a similar advantage to our approach in that it can be used by developers unfamiliar with the system since it employs an existing test suite. DFTs require partitioning of the test suite whereas FS requires two version of the system in between which a feature was introduced or changed. DFTs and FS are complementary approaches that are applicable under different circumstances.

Reps et al [10] use spectral analysis to help look for Y2K bugs. They feed a pre-millennium date and a post-millennium date into the same system. Divergences in the execution profile are the first place they look for flaws in the code. Their use of good and bad dates is analogous to correlated exhibiting and non-exhibiting test sets.

7. Summary

We introduced the concept of Dynamic Feature Traces (DFTs) which assist developers in feature location. We have described the process to create DFTs. Part of this process uses heuristics to rank methods in relation to what extent a method is relevant to a feature. We claim that DFTs are more effective than similar techniques when applied to a test-suite partitioning created with moderate effort by a developer unfamiliar with the target system.

To substantiate the claim, we described an experiment that compares DFT creation to the results of a binary technique that determines UCOMPs. We show that for the feature enhancements analyzed, the UCOMPs located by the binary technique are not as effective as DFTs are for finding relevant code.

We also described a case study we performed using DFTs to apply a feature enhancement to an existing system. This study provided anecdotal evidence that DFTs may be useful for feature enhancement tasks.

References

[1] D. Astels. *Test Driven Development: A Practical Guide*. Prentice Hall PTR, first edition, Aug. 2003.

[2] K. Beck. *Test Driven Development*. Addison-Wesley Pub Co, first edition, Nov. 2002.

[3] P. Cederqvist. *The CVS manual — Version Management with CVS*. Network Theory Ltd., first edition, Dec. 2002.

[4] K. Chen and V. Rajlich. Case study of feature location using dependence graph. In *Proceedings of the 8th International Workshop on Program Comprehension*, pages 241–247. IEEE Computer Society, 2000.

[5] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *IEEE Trans. Softw. Eng.*, 29(3):210–224, 2003.

[6] D. Janzen and K. De Volder. Navigating and querying code without getting lost. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 178–187. ACM Press, 2003.

[7] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353. Springer-Verlag, 2001.

[8] D. R. Licata, C. D. Harris, and S. Krishnamurthi. The feature signatures of evolving programs. In *Proceedings of the* 18th *International Conference on Automated Software Engineering*, pages 281–286. IEEE Computer Society, Oct. 2003.

[9] V. Massol and T. Husted. *JUnit in Action*. Manning Publications Co., first edition, Nov. 2003.

[10] T. Reps, T. Ball, M. Das, and J. Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In M. Jazayeri and H. Schauer, editors, *Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97)*, pages 432–449. Springer–Verlag, 1997.

[11] M. P. Robillard and G. C. Murphy. Concern graphs: finding and describing concerns using structural program dependencies. In *Proceedings of the 24th international conference on Software engineering*, pages 406–416. ACM Press, 2002.

[12] M. P. Robillard and G. C. Murphy. Automatically inferring concern code from program investigation activities. In *Proceedings of the* 18th *International Conference on Automated Software Engineering*, pages 225–234. IEEE Computer Society, Oct. 2003.

[13] N. Wilde and C. Casey. Early field experience with the software reconnaissance technique for program comprehension. In *Proceedings of the 1996 International Conference on Software Maintenance*, pages 312–318. IEEE Computer Society, 1996.

[14] N. Wilde and M. Scully. Journal of software reconnaissance: Mapping features to code. *Software Maintenance: Research and Practice*, 7(1):49–62, 1995.

[15] W. E. Wong, J. R. Horgan, S. S. Gokhale, and K. S. Trivedi. Locating program features using execution slices. In *IEEE Symposium on Application-Specific Systems and Software Engineering*, pages 194–203, Mar. 1999.

[16] S. S. Yau, R. A. Nicholl, J. J.-P. Tsai, and S.-S. Liu. An integrated life-cycle model for software maintenance. *IEEE Trans. Softw. Eng.*, 14(8):1128–1144, 1988.

[17] W. Zhao, L. Zhang, Y. Liu, J. Sun, and F. Yang. SNIAFL: Towards a static non-interactive approach to feature location. In *Proceedings of the 26th International Conference on Software Engineering (ICSE'04)*, pages 293–303. IEEE Computer Society, 2004.