CERBERUS: Tracing Requirements to Source Code Using Information Retrieval, Dynamic Analysis, and Program Analysis

Marc Eaddy, Alfred V. Aho
Department of Computer Science
Columbia University
New York, New York, USA
{eaddy, aho}@cs.columbia.edu

Giuliano Antoniol
Département de Génie Informatique
École Polytechnique de Montréal
Montréal, Québec, Canada
antoniol@ieee.org

Yann-Gaël Guéhéneuc
Département d'Informatique et Recherché Opérationnelle
Université de Montréal
Montréal, Québec, Canada
guehene@iro.umontreal.ca

Abstract

The concern location problem is to identify the source code within a program related to the features, requirements, or other concerns of the program. This problem is central to program development and maintenance. We present a new technique called prune dependency analysis *that can be combined with existing techniques to dramatically improve the accuracy of concern location. We developed* CERBERUS*, a potent hybrid technique for concern location that combines* information retrieval*,* execution tracing*, and* prune dependency analysis*. We used* CERBERUS *to trace the 360 requirements of* RHINO*, a 32,134 line Java program that implements the ECMAScript international standard. In our experiment, prune dependency analysis boosted the recall of information retrieval by 155% and execution tracing by 104%. Moreover, we show that our combined technique outperformed the other techniques when run individually or in pairs.*

1. Introduction

Most of the cost in software development goes towards maintaining and evolving existing code [5]. Programmers spend more than half of their time trying to understand the program so that they can make changes correctly [26]. The most valuable information to a programmer—and the most difficult to obtain—is the *"intent behind existing code and code yet to be written"* [17]. To understand the intent behind a piece of unfamiliar code, the programmer may look for clues in the code or documentation. Program identifiers and comments, programming patterns and idioms, program dependencies, and how the program responds to input, all hint at the relationship between the code and the features, requirements, and other high-level concerns of the program.

A *concern* is *any consideration that can impact the implementation of a program* [21]. Software requirements and features (i.e., user-observable functionality) are examples of different types of concerns Generally speaking, the goal of *concern location* (e.g., concept assignment [4], feature location [2] [8] [12] [13] [18] [20] [25] [27] [28] [29], requirements tracing [1] [9] [10] [15], and related techniques) is to determine the relationship between high-level domain concepts (concerns) and other software artifacts, e.g., source code.

A *concern location technique* can be thought of as an *expert* that evaluates a set of *heuristics* to produce a *relevance judgment* about the relationship between a program element and a concern. For example, information-retrieval-based techniques are experts that provide judgments based on the lexical similarity between program element names and concern descriptions. Unfortunately, many factors conspire to reduce the reliability of the judgments. First, each expert relies on the presence of clues to evaluate its heuristics. If the clues are missing or misleading (e.g., meaningful element names are not used), the expert will likely draw the wrong conclusions. Second, each expert is focused on a narrow aspect, and thus, an incomplete picture, of the program. The judgments produced by one expert may not be entirely trustworthy.

In theory, we should be able to increase accuracy by combining the judgments of multiple experts, thus mimicking how a programmer combines multiple clues to arrive at a more informed judgment. That is, if three independent experts arrive at the conclusion that an element is relevant to a concern, this judgment is more reliable than if only one or two experts agreed. However, combining judgments is not straightforward because of disagreements between the experts. Furthermore, judgments are not necessarily comparable because each expert analyzes data and produces judgments in a different way (i.e., impedance mismatch).

Our goals in this paper are to introduce improvements to previous automated concern location techniques, to present a novel prune dependency analysis technique, and to determine if the judgments of three experts can be successfully combined—would it be a case of "three heads are better than one" or "too many cooks in the kitchen?" Our hypothesis is that combining the judgments of all three

experts—information retrieval, execution tracing, and prune dependency analysis—is better than combining one or two experts. We evaluated our hybrid technique, CERBERUS[1], by comparing it against a large concern–element mapping (10,613 concern–element links) done by hand for RHINO, a 32,134 line Java program. While a handful of researchers [18] [20] [28] [29] have successfully paired concern location techniques, as yet no one has combined the judgments of three experts.

We found that concern location accuracy can improve as more heuristics and analysis techniques are brought to bear. In particular, CERBERUS outperforms all other technique combinations by 9%–263%. In large part this improvement is due to the use of our novel prune dependency analysis technique, which boosted the recall of information retrieval by 155% and execution tracing by 104%.

In the next section we give an overview of the individual techniques and how we integrated them to create CERBERUS. In Section 3, we describe our evaluation methodology and our case study. We describe our experimental results in Section 4 and evaluate the different technique combinations, including CERBERUS. In Section 5, we discuss some surprising results and threats to validity. Section 6 reviews prior work. Section 7 concludes.

2. An Overview of Our Approach

We are interested in determining if a *prune dependency* relationship exists between a concern and an element based on the following rule [9]:

> *A program element is relevant to a concern if it should be* ***removed****, or otherwise* ***altered****, when the concern is pruned.* [2]

To properly interpret this rule, consider a *software pruning* scenario where a programmer is removing a concern to reduce the footprint of a program or otherwise tailoring the program for a particular environment. Thus, the goal is to remove as much of the code related to a concern as possible, short of a redesign[3], and without affecting other concerns.

To achieve this goal, the programmer must locate as many elements in the program as possible that implement the concern. Locating these elements by hand is labor intensive and does not scale to large constantly evolving programs. Fortunately, she can use automated concern location techniques. Next, we describe two well-known concern location techniques and introduce our novel prune dependency analysis technique.

2.1. Information Retrieval

Information-retrieval-based concern location techniques (IR), such as vector space indexing [1] [28] [29], probabilistic indexing [1], and latent semantic indexing (LSI) [1] [18] [20], have been very successful at determining relevance based on the similarity between terms used in the concern descriptions and in the program elements (e.g., element names, variable names). Unfortunately, IR techniques suffer from the same problems that impede information retrieval in the large, such as polysemy (different meanings for the same word) and synonymy (same meaning for different words). These problems particularly impact concern location because concern descriptions (e.g., paragraphs from a requirements specification) and program elements are often written by different authors, using different vocabularies and rules of grammar [20], and because program identifiers are often abbreviated (e.g., "num," "str"). IR techniques require meaningful identifiers to be used to establish relevancy with any certainty.

Our IR technique parses Java source code and extracts terms from comments and identifiers. We include comments that precede the element definition as well as comments inside methods. Identifiers include the names of methods (and fields), enclosing types, packages, parameters, local variables, field accesses, method calls, and type references (e.g., using *instanceof*). Qualified names (e.g., *foo.bar.baz()*) are split into separate identifiers ("foo," "bar," and "baz"). Compound identifiers are added as is and also split into separate terms based on standard naming conventions (e.g., "camelCaseStyle" and "underscore_style"). We kept terms that looked like section numbers (e.g., "15.4.4.5") as well as "magic" numbers (e.g., 0, 1, 3.14159).

We filter terms using a standard stop list augmented with the list of Java keywords. Abbreviated identifiers were generating a large number of synonyms, so we created a custom thesaurus to expand them. A unique aspect of our technique is that we also expand abbreviated compound identifiers, e.g., the identifier "numConns" expands into terms "numberconnections," "number," and "connections." To further reduce the number of unique terms, we reduced the inflected form of the terms to their stem using a standard stemming library.

Similar to [29], we treat program elements as the "documents" and requirements as the "queries." Our custom script parses a requirements specification to identify individual requirements, which will become the queries. Terms are extracted from the requirements using the same rules as above for program elements.

After extracting all the terms for the program elements and requirements, we weigh the terms using the standard

[1] The three-headed dog of Greek mythology.

[2] "Prune" is synonymous with "remove." We use different terms to make it clear *what* is being removed: the concern (*pruned*) or the program element (*removed*).

[3] Assume that disabling the concern using a flag, preprocessor macros, or code generation is not sufficient.

TF–IDF formula [3]. The result is a vector of weighted terms for each program element and requirement. We calculate the *cosine distance* [3] between the two vectors to arrive at a *similarity score* that ranges from 0 (no terms in common) to 1 (terms are identical). IR presumes that term similarity implies relevance, so we interpret the similarity score to be the *predicted relevance score* for a given program element and requirement.

2.2. Execution Tracing

Execution-tracing-based concern location (*tracing*) techniques analyze the runtime behavior of a program to determine program elements *activated* when a concern is *exercised*. For functions and methods, activation implies *execution*; for fields and global variables, activation implies *access* (reading or writing). Exercising a concern implies supplying inputs to the program that are related to the concern. For example, the "bookmarking" feature in a web browser can be exercised by clicking the "Add to Bookmarks" button [20]. We improve upon previous execution tracing techniques by considering field accesses as well as method executions.

Traces for a set of concerns are compared to distinguish elements that are specific to a particular concern from those shared by many concerns. The output is a list of methods ranked by their *trace score* (predicted relevance score), which ranges from 0 (irrelevant) to 1 (relevant). We evaluated the effectiveness of the following scoring formulas:

- *Software Reconnaissance (SR)*: 1 if the element if activated by only one concern, otherwise 0 [25].
- *Dynamic Feature Traces (DFT)*: *# element activations by the concern / total # element activations* [2].
- *Scenario-Based Probabilistic Ranking (SPR)*: *# of tests for a concern that activate the element / total # of tests that activate the element* [13].
- *Element Frequency–Inverse Concern Frequency (EF–ICF)*:

$$EF\text{–}ICF = \frac{\#\ element\ activations\ by\ the\ concern}{total\ \#\ element\ activations} \times \log\left(\frac{\#\ concerns\ that\ activate\ any\ element}{\#\ concerns\ that\ activate\ the\ current\ element}\right)$$

The first scoring formula, introduced by Wilde and Scully in their pioneering work on Software Reconnaissance [25], renders a binary judgment based on whether an element is uniquely activated by a concern. Later formulas (e.g., DFT, SPR, EF–ICF) more closely capture the notion of *element specificity*, i.e., the *degree* to which an element is relevant to a concern, and at the same time compensate for the imprecision and noise inherent in the collection of dynamic data [2].

We created the last formula, *Element Frequency–Inverse Concern Frequency (EF–ICF)*, because we noticed the resemblance between DFT and SPR and the TF–IDF-like [3] term weighting formula commonly used in information retrieval. EF–ICF is the same as TF–IDF, where "terms" are program elements, "term frequencies" are activation counts (e.g., number of method calls), and "documents" are concerns. EF–ICF essentially adjusts the Dynamic Feature Traces score to account for the likelihood that the element is activated by other concerns.

2.3. Prune Dependency Analysis

Given an initial set of elements relevant to a concern (*initial relevant elements* or *seeds* [28] [29]), program-analysis-based concern location techniques infer *additional relevant elements* by analyzing different kinds of relationships between the elements. The initial seeds may be found by manual inspection [16], or, as in the case of CERBERUS, by a separate automated concern location technique. We say an element e_c is relevant to concern c if e_c is *prune dependent on c*; that is, pruning c implies e_c should be *removed* from the program, or otherwise *altered*. Our *prune dependency analysis* (PDA) technique assumes all the initial relevant elements will be removed when c is pruned. PDA then determines the impact on the program of removing the initial relevant elements to infer additional relevant elements.

PDA takes as input a tuple (G, R, A). G is a directed program dependency graph $G = (E, D_1, D_2, D_3)$, where E is the set of program elements, D_1 are *references* dependencies (e.g., method *A calls* method *B*), D_2 are *contains* dependencies (e.g., a class *contains* its members), and D_3 are *inherits-from* dependencies. $R = \{e \mid e \in E\}$ is called the *removal set* and initially contains the initial relevant elements for a single concern c. $A = \{e \mid e \in E\}$ is called the *alter set* and is initially empty.

The dependency graph for the example Java program in Fig. 1 is shown in Fig. 2a. Elements in the removal and alter sets are indicated as dark grey and light grey nodes, respectively. The PDA algorithm proceeds by analyzing each element in the removal set R to determine if removing

```
interface A {
    public void foo();
}
public class B implements A {
    public void foo() { ... }   } Initial
    public void bar() { ... }   } relevant elements
}
public class C {
    public static void main() {
        B b = new B();
        b.bar();
    }
}
```

Fig 1. Example to illustrate prune dependency analysis

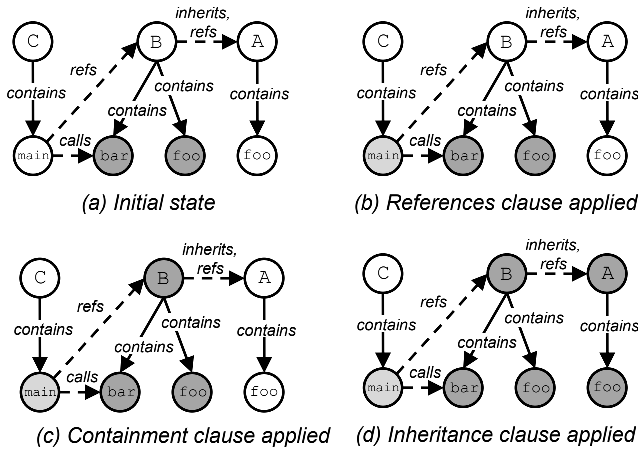

Fig 2. Application of prune dependency analysis

that element will require additional elements to be removed—in which case they will be added to *R*—or otherwise altered—in which case they will be added to *A*. The following clauses were derived from [9].

References clause – If element *e references* e_c, then *e* is also relevant to *c* since removing e_c requires all references to it in the code to be modified to avoid a compile error. Depending on whether *e* is a method, field, or type (class, interface, or enum), "references" means *calls*, *accesses*, or *names*, respectively. A type is referred to by name in Java by type casts, *instanceof*, *extends*, and *implements*.

The removal set in Fig. 2a initially contains *B.foo()* and *B.bar()*. This means that removing *c* dictates that these methods be removed. However, *C.main() calls B.bar()*, so we must *alter C.main()* to avoid a compile error, e.g., remove the call entirely or replace it with something else. Thus, *C.main()* is added to the alter set, as shown in Fig. 2b.

Element containment clause – If all elements *contained by e* are relevant to *c*, then *e* is also relevant to *c*. Referring to Fig. 2b, pruning *c* implies all the methods of *B* will be removed, leaving behind an empty class. Adhering to the goal of software pruning, we should remove *B* as well. Thus, *B* is added to the removal set, as shown in Fig. 2c.

Inheritance clause – If all subtypes *inherited from* an abstract base class or interface *e* are relevant to *c*, then *e* is also relevant to *c*. Continuing the example, removing *B* will leave *A* without a concrete implementation. Adhering to the goal of software pruning, we should remove *A* since an object of type *A* or derived from *A* cannot be created and methods of *A* cannot be called. Thus, *A* and its members are added to the removal set, as shown in Fig. 2d. We reapply the clauses to *A* and its members but this does not affect the removal set, so the algorithm terminates.

In this way, PDA builds up the set of elements that must be removed or altered when the concern is pruned. The algorithm terminates when all elements in *R* have been analyzed, which is guaranteed to happen since the elements are finite and are only analyzed once. PDA outputs the final set of removal and alter elements. The *final relevant elements* are the union of the two sets, i.e., $R \cup A$. For our example, the final relevant elements are *A*, *A.foo()*, *B*, *B.foo()*, *B.bar()*, and *C.main()*.

PDA assumes a closed world where the types are known at analysis time. This is a reasonable assumption since our goal is to locate concerns in our own code.

2.4. Our Combined Technique: CERBERUS

As shown in Fig. 3, CERBERUS combines information retrieval, execution tracing, and prune dependency analysis to locate concerns. We use information retrieval to produce the IR ranking and execution tracing to produce the trace ranking. We use the PROMESIR formula [20] to combine the rankings produced by tracing and IR by normalizing, weighting, and summing the similarity and trace scores. To be conservative, absent scores, i.e., an element that does not appear in a trace, are treated as 0 (irrelevant).

We then apply a threshold to identify highly relevant elements, which will become *seeds* for the prune dependency analysis technique. We experimented with rank thresholds (e.g., the top 5 most similar elements), score thresholds (e.g., the elements with scores in the top 5%), and the *gap threshold algorithm* introduced by Zhao

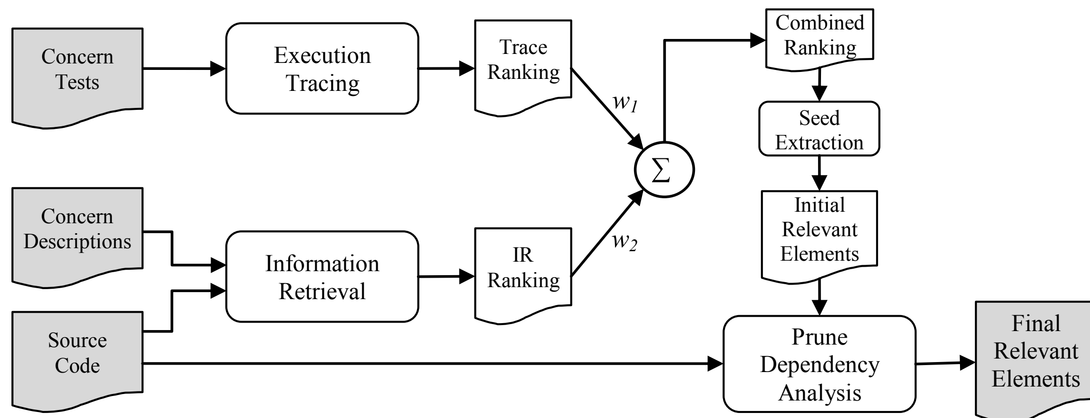

Fig 3. Overview of the Cerberus approach

et al. [28] The gap algorithm traverses the list of elements in order, from highest to lowest score, to find the widest gap between adjacent scores. The relevance score immediately prior to the gap becomes the threshold for picking the seeds. These seeds are then passed to the prune dependency analysis to infer additional relevant elements.

3. Evaluation Methodology

The goals of our experiment are to determine if combining information retrieval, execution tracing, and program dependency analysis improves accuracy over individual and paired techniques, and to determine if our prune dependency analysis technique is effective at inferring relevant elements.

3.1. Effectiveness Measures

We use the key measures of information retrieval, *precision* and *recall*, to evaluate concern location techniques [15]. For a given concern *c*, let E_c be the set of program elements actually relevant to *c* (the *relevant element set*), and E_c^* be the set of elements *judged* to be relevant by our technique (the *retrieved element set*). When the concern location technique produces a list of elements e_i ranked by a relevance score, we apply a threshold *t* to obtain the retrieved element set by discarding elements below the threshold [2].

Recall is the percentage of relevant elements retrieved by the technique, i.e., *recall* = $|E_c \cap E_c^*|/|E_c|$. *Precision* is the percentage of retrieved elements that are relevant, i.e., *precision* = $|E_c \cap E_c^*|/|E_c^*|$. We also compute the *f-measure*, the harmonic mean of precision and recall: *f-measure* = 2 · (*recall* · *precision*) / (*recall* + *precision*). The f-measure facilitates comparing effectiveness because it produces a single value that balances precision and recall equally.

We are interested in the overall accuracy of a technique for multiple concerns, so we compute *mean recall*, *mean precision*, and *mean f-measure*. However, a concern location technique may not locate all the concerns. For example, execution tracing can only locate concerns that have an associated exercising test. It is also possible that all of the elements in a ranking are below the relevance score threshold. We do not consider missing concerns when computing means for several reasons. First, precision and f-measure are undefined in this case. Second, our graphs show the number of concerns located by the techniques. Third, we are interested in how well a technique performs in two different scenarios: when we desire a trace of all the concerns (i.e., requirements traceability) and when we want to locate a subset of concerns (i.e., concern location). As we will see, we can achieve higher accuracy if we can accept locating fewer concerns.

We say that a ranking technique is *effective* if higher relevance scores lead to higher *actual relevance*; that is, relevance scores should be positively correlated with f-measure. We say that a technique is more effective than another if it has a higher mean f-measure. Because dependency analysis requires preexisting seeds, we did not evaluate it in isolation, but instead evaluated it in combination with execution tracing and information retrieval. We consider prune dependency analysis to be effective if it increases the f-measure of another technique.

3.2. Our Study Subject: RHINO

RHINO[4] is a JavaScript/ECMAScript interpreter and compiler. Version 1.5R6 consists of 32,134 source lines of Java code (excluding comments and blank lines), 138 types (classes, interfaces, and enums), 1,870 methods, and 1,339 fields (as reported by CONCERNTAGGER[5]). RHINO implements the ECMAScript international standard, *ECMA-262 v3* [11]. We chose RHINO because in a prior study [10] we had systematically associated all the methods, fields, and types in the RHINO source code with the requirements from the ECMAScript specification, which provides us with an oracle by which we can evaluate the performance of different concern location techniques. Indeed, the extensive manual labor required to map RHINO—it took 102 hours to create 10,613 concern–element links—strongly motivated our current work.

We evaluated our technique against the same set of concerns analyzed in our prior study, namely the requirement concerns specified by the normative leaf sections of the ECMAScript specification [11]. A *leaf section* is a section that has no subsections; i.e., it is at the lowest level of granularity. For example, consider the following snippet of the requirements hierarchy:

15 – Native ECMAScript Objects
...
15.4 – Array Objects
...
15.4.4 – Properties of the Array Prototype Object
...
15.4.4.5 – Array.join(separator)
...

In the above snippet, "15.4.4.5 – Array.join()" is a leaf section. The ECMAScript requirements are quite detailed, as can be seen from an excerpt for this section:

> *"The elements of the array are converted to strings, and these strings are then concatenated, separated by occurrences of the* separator*. If no separator is provided, a single comma is used as the separator. The join method takes one argument,* separator*, and performs the following steps: ... [algorithm follows]"*

[4] http://www.mozilla.org/rhino

[5] http://www.cs.columbia.edu/~eaddy/concerntagger

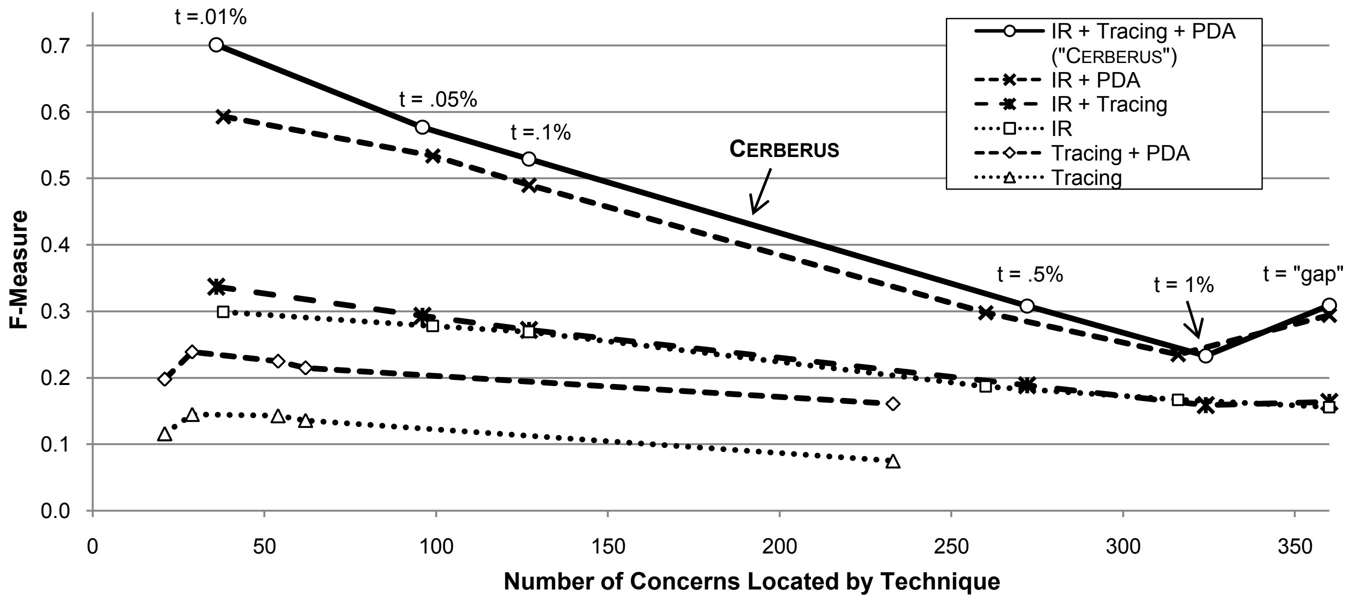

Fig 4. Effectiveness of individual concern location techniques and technique combinations for different threshold values *t*

The top four TF–IDF weighted terms for this section are "15.4.4.5," "comma," "join," and "occur."

3.3. Applying CERBERUS to RHINO

Indexing the RHINO source code resulted in 4,530 unique terms (after stopping, stemming, compound term expansion, and abbreviation expansion) and 3,345 documents, i.e., one document for every type, method, and field in RHINO. Indexing the ECMA 262 v3 specification resulted in 2,032 unique terms and 360 leaf sections, which became our queries. The total vocabulary size (combining code and requirements terms) was 5,566 unique terms.

The ECMAScript test suite consists of 939 tests, 795 of which directly test 240 of the 360 requirements, i.e., several requirements have multiple tests. These tests activate 1,124 of the 1,870 methods in RHINO. Thus, the concern coverage of the test suite is 67% and the method coverage is 60%. The tests are written in ECMAScript and the file names indicate which requirement is tested (e.g., "15.4.4.5.js" tests section "15.4.4.5 – Array.join())."

We collected traces of method calls and field accesses using AspectJ[6]. AspectJ is not able to monitor access to "static final" fields because Java compilers convert these into constants. Therefore, when we are combining the trace and IR scores, we just use the IR score instead of assuming the trace score is 0 as we do for other missing elements (see Section 2.4).

Similar to [18], we semi-automatically inserted method calls into the test cases to start and stop tracing to avoid tracing startup code, setup code, test harness code, and code related to other concerns. The assumption is that this code is irrelevant to the concern and acts as noise that reduces the ability for the traces to discriminate relevant elements. This assumption later proved false for reasons we explain in Section 5.

We extended our CONCERNTAGGER plug-in for ECLIPSE[7] to implement prune dependency analysis by leveraging ECLIPSE's program analysis framework.

4. Analysis of Experimental Results

Fig. 4 shows a graph of the mean f-measure for all combinations of the three techniques. Each line in the graph represents a different technique or technique combination. Each point on a line represents a different threshold *t*. Each technique is represented in its optimal configuration, i.e., parameterized to use the most effective heuristic. For example, the best parameterization of the tracing technique used marked traces and the Dynamic Feature Traces scoring formula. In Fig. 6, we show how the other scoring formulas fared.

4.1. Comparing Technique Effectiveness

We make several observations:

The combination of the three techniques is the most effective. From Fig. 4, we see that our combined technique, CERBERUS (IR + Tracing + PDA), is more effective than any subset.

We verified that CERBERUS outperforms the other techniques by statistical analysis of the populations of the f-measures produced by the compared techniques. We

[6] http://www.aspectj.org

[7] http://www.eclipse.org

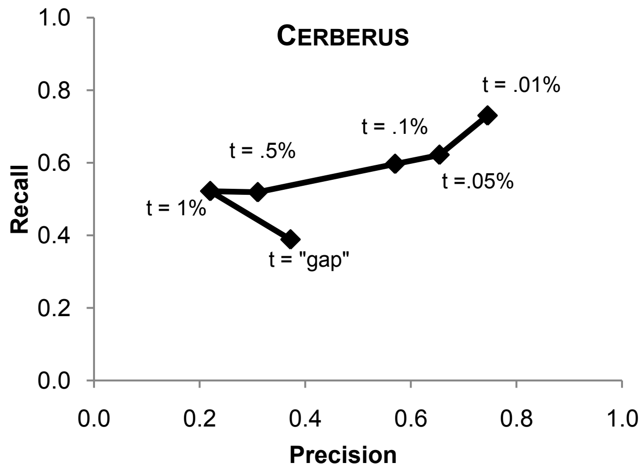

Fig 5. Precision–recall graph for our combined technique

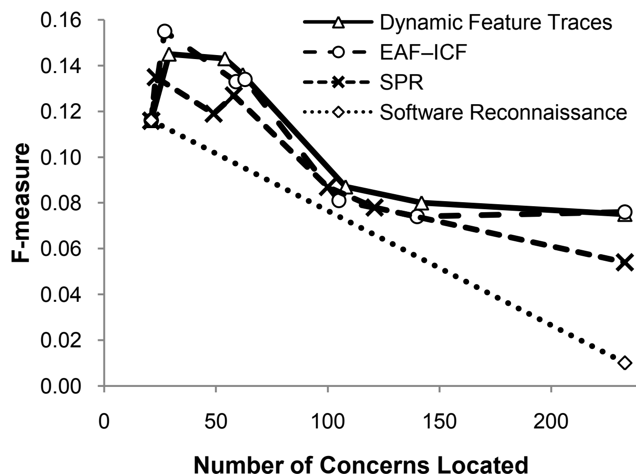

Fig 6. Effectiveness of execution tracing for different trace relevance scoring formulas

observed that the f-measure values for the techniques were not normally distributed. We therefore performed one descriptive statistic test and three non-parametric tests: mean f-measure, Mann–Whitney, Kruskal–Wallis, and Friedman [24]. For the first test, we simply computed the mean f-measure value for each technique. The non-parameteric tests compare the f-measure ranks rather than the actual values. All four tests indicate that the mean f-measure values and ranks are highest for CERBERUS, which supports our performance claim according to the effectiveness criteria in Section 3.1. The Kruskal–Wallis and Friedman tests further determined that the difference in means was statistically significant (p-value < .005). However, while we witnessed an average 9% improvement in f-measure for CERBERUS over IR + PDA, as can be seen in Fig. 4, the Mann–Whitney test was inconclusive since it indicated the improvement was not statistically significant (p-value = .291). In short, three tests indicate the superiority of CERBERUS, while one test was inconclusive.

The PROMESIR formula allows us to adjust the weights of the IR and trace relevance scores to reflect our confidence in these two “experts.” However, weighting the experts equally, as done in [20], produces worse results than using IR alone, because in our study this would be akin to averaging the test scores of a poorly performing student (tracing) and a well performing student (IR). Using an 80/20 combination (determined by trial-and-error) of IR and trace scores, execution tracing improves the performance of IR by 4% and the performance of IR + PDA by 9%.

Fig. 5 shows the precision–recall graph for our approach, where each point on the line represents a different threshold t. Our best precision is 75% and recall is 73% using the threshold $t = 0.01\%$; however, looking at Fig. 4 we see that only 36 concerns have ranked elements that exceeded this threshold. If we use the “gap” threshold, we locate all 360 concerns, but our precision drops to 37% and recall drops to 39%.

No previous study that we know of has traced this many concerns (360) to the level of methods and fields in a program of this size (32,134 lines). Indeed, the existence of a publicly available oracle[8] of this size (10,613 concern–element links) is rare. While our precision and recall (48% and 56%, respectively) is not as good as SNIAFL [29] (99% and 87%), for example, our evaluation is more rigorous (360 concerns versus 21) and realistic (32.1K lines versus 8.6K lines).

Each technique and technique combination is effective at locating concerns. In general, higher relevance scores lead to higher actual relevance. This is apparent by noticing that the f-measure tends to increase as we increase our relevance score threshold. This is not surprising, as the IR and execution tracing techniques have been independently validated by numerous studies. However, only a few studies [28] [29] have validated the effectiveness of using program analysis for automated concern location, and our results provide further evidence.

Prune dependency analysis is effective at boosting the performance of other techniques. Combining prune dependency analysis with any other technique or technique combination improves the recall of that technique, regardless of the threshold or number of concerns located. This can be observed in Fig. 4, by noting that combining PDA with any technique results in a higher f-measure.

Software Reconnaissance is not as effective as the other trace scoring formulas. The graph in Fig. 6 indicates that the Software Reconnaissance formula performs poorly, possibly validating the claim in [13] that this formula is *“overly sensitive to the quality of the input.”* SPR performed better followed by DFT and EF–ICF, although we did not find a significant difference between these last two.

[8] http://www.cs.columbia.edu/~eaddy/concerntagger

TABLE 1
TF–IDF Similarity Scores for Concern
"15.4.4.5 – Array.join()"

Rank	*Program Element*	*Kind*	*Similarity Score*
1	NativeArray.Id_join	field	.22
2	NativeArray.js_sort()	method	.22
3	NativeArray.js_join()	method	.21
	– Largest Similarity Score Gap –		
4	NativeArray.Id_concat	field	.18

4.2. A Concern in Point

The following example illustrates how prune dependency analysis can improve the recall of information retrieval. The Rhino concern "15.4.4.5 – Array.join()" specifies the behavior of the *join()* method of the native ECMAScript Array class. Table 1 shows the results of our IR technique.

The similarity gap between elements ranked #3 and #4 is .21-.18 = .03, which is the largest gap in the ranking, so the threshold t is .21, and only the first three elements are judged relevant. *Id_join* and *js_join()* both use the terms "array" and "join," which also occur in the concern description. The relatively low frequency of the term "join" in the specification results in a high TF–IDF weight, as we showed in Section 3.2. Unfortunately, *js_sort()* is actually *irrelevant* but receives a high similarity score because of an incorrect comment in the code.

From the bad *js_sort()* seed, PDA infers the following methods in the NativeArray class: *execIdCall()*, *findPrototypeId()*, and *initPrototypeId()*. Coincidentally, the good *js_join()* seed infers the same methods, and these inferred methods are in fact relevant, so the damage of the bad seed is masked. Thus, the IR + PDA technique finds all the relevant elements (recall = 100%) and one irrelevant element (precision = 83%).

5. Discussion

5.1. Poor Performance of LSI and Tracing

We were surprised to find that LSI did not improve performance over vector space indexing despite the success that others researchers have had with it (e.g., [1] [18] [20]). One possible reason is that our corpus is somewhat small, consisting of 5,566 unique terms and 3,345 documents, thus reducing the effectiveness of LSI. For example, LSI is effective at reducing the problems associated with term synonymy by analyzing the co-occurrence of terms in the corpus. If the corpus is small, LSI may not have enough co-occurrence information to be effective.

We were also surprised that the effectiveness of our execution tracing technique could not be improved, even after instrumenting the 939 tests in the RHINO test suite with trace markers, employing two different trace tools, and leveraging an arsenal of different scoring formulas. There are several factors that may account for this poor performance.

The first issue is that 33% of the concerns in RHINO are untested, which limits effectiveness when combining tracing with other techniques because tracing provides no information for those missing concerns. 40% of the methods in RHINO are not called during any test, although some were actually relevant to some of the concerns.

The second issue is that some concerns were implemented as different branches of a *switch* statement or *if* statement as opposed to different methods. Our scoring formulas deem a method with many concern-relevant branches as being irrelevant to any concern in particular, when in fact the opposite is true. We believe that tracing at the basic-block level [27] [12], or emulating such a trace, would be effective at untangling these interleaved concerns.

The third issue is that execution trace data tends to be imprecise and noisy because it includes startup code, test harness code, bookkeeping code, etc., which are not relevant to the concern [2] [12] [13]. Imprecision and noise hinders establishing the non-exhibiting scenarios that help pinpoint relevant methods. These problems made it difficult to use RHINO's tests to isolate concern-relevant code. Liu et al. [18] argued that information retrieval can compensate for this issue, but our results indicate that a test suite with poor isolation cannot be effectively repurposed for concern location.

We attempted to isolate concerns by bracketing the functionality of the primary concern in each test with trace markers, but this did not improve results. Indeed, the markers ended up excluding many relevant methods and fields, e.g., concern-relevant initialization code. Overall accuracy was mostly unaffected by the use of trace markers: analysis of the f-measures for the different thresholds revealed that marked traces increased the f-measure by an insignificant 0.01.

Interestingly, we achieved a greater performance improvement by using trace scoring formulas that attempt to compensate for imprecision and noise than by using trace markers. This is analogous to the classic information retrieval debate that a stop list is redundant if we have a good TF–IDF formula trained on a large corpus.

Note that the poor performance of tracing actually validates our approach since combining expert judgments reduces the impact of "unqualified experts." By choosing appropriate relevance thresholds and confidence weights, we were able to ignore bad judgments produced by tracing while retaining good ones, to boost the accuracy over IR by 4% and over IR + PDA by 9%. Furthermore, Fig. 4 suggests that CERBERUS is effective even without execution tracing.

5.2. Threats to Validity

Internal validity deals with possible biases and measurement errors. Our current study is almost identical to our prior study [10], e.g., same subject program, concerns, and assignment criteria. The only difference is that our current study automatically locates concerns whereas the previous study located concerns manually. Because our oracle was created for the prior study, *subject bias* is not an issue; i.e., the author could not craft the oracle to maximize the results of our automated technique.

To avoid the possibility that a tool or technique is defective, we used multiple tools and techniques to redundantly verify some of our results. Because we implemented execution tracing using two different instrumentation tools, and trace markers were manually added to the tests by two authors independently, we cannot attribute the poor performance of tracing to defective tools. We also evaluated two different information retrieval techniques: latent semantic indexing and vector space indexing. The results of the two techniques were very similar, and thus redundantly indicate the maximum effectiveness we can expect from an information-retrieval-based technique for tracing concerns in RHINO.

External validity is the extent to which we can generalize our findings to other contexts. Concern location techniques reflect the assumptions and goals of their authors. Although the concern–code relationship induced by our prune dependency rule was demonstrated to be useful for assessing code quality, predicting defective concerns [10], and requirements tracing and validation, the rule may need to be specialized to perform well for other software engineering tasks. For example, concern-relevant fields may not be useful when predicting the locations of defects.

6. Related Work

In comparison to other information-retrieval-based concern location techniques, CERBERUS is the first to use a thesaurus to expand abbreviated terms and to index fields as well as methods, thus providing a more complete concern–code mapping.

In contrast to Dynamic Feature Traces [13], our Scenario-based Probabilistic Ranking heuristic calculates a relevance score based on the probability that an element is activated by a concern [2], analyzes field accesses as well as method calls, and uses marked traces [18] to ensure that only concern-related functionality is traced.

Several researchers [4] [8] [22] have exploited program dependency heuristics for interactive concern location, but only a few have used them for automated concern location. Zhao et al.'s [28] *complementation heuristic* assumes that any method that transitively calls, or is transitively called by, a relevant element is also relevant. They use *branch-reserving call graphs* in [29] to generate *pseudo-execution traces*, essentially using inexpensive static analysis to emulate the SR heuristic. PDA's *references* clause infers almost the same elements, except that we always consider a referencing element relevant even if it is already relevant to another concern. Moreover, the *contains* and *inherits* clauses are unique to PDA.

Recently, researchers have combined heuristics to improve accuracy. For example, information retrieval heuristics have been combined with execution heuristics [18] [20] and program dependency heuristics [28] [29]. However, CERBERUS is the first to combine all three.

There are many different ways to integrate execution trace data when creating a combined technique. CERBERUS converts the trace data into a *ranking* and then combines it with an IR ranking. In contrast, SITIR [18] uses the trace data to *filter* the input to IR—the heuristic being that methods not activated when a concern is exercised cannot be relevant to that concern [29], and should therefore be ignored during the information retrieval phase. The drawback of using tracing as a filter is that it is sensitive to the capability of the tests to isolate concerns, which may be quite poor [13]. Indeed, when we experimented with using the RHINO trace data as a filter we found that a large number of relevant methods were filtered due to the issues we described in Section 5.2. By using the trace data to rank instead of filter elements, we were able to adjust the influence of the trace data, and thereby compensate somewhat for the poor test suite quality.

DORA [16] is complementary to CERBERUS in that DORA could consume the seeds that CERBERUS produces, thus eliminating user involvement. Alternatively, CERBERUS could use IR to filter the final relevant element set.

Essentially, CERBERUS is a marriage of the PROMESIR and SNIAFL approaches, with several improvements. Whereas PROMESIR is effective at combining IR and tracing to pick highly relevant seed elements, SNIAFL is effective at using program analysis to infer additional relevant elements from those seeds. We refined these approaches by adding a custom thesaurus, using marked traces, employing the prune dependency heuristic, and considering fields as well as methods.

7. Conclusions and Future Work

CERBERUS is the first concern location technique to combine information retrieval, executing tracing, and program analysis to locate concerns in source code. We showed that the combination of these three analysis methods produced more accurate results than all other combinations of these methods. Our novel prune dependency analysis algorithm boosted the recall of information retrieval by 155% and execution tracing by 104%, with only a negligible loss (0.5%) in precision.

We believe hybrid concern location techniques, such as ours, which combine multiple analyses, will become a major new avenue of research. Analyses that incorporate

more information, such as line co-change [7], conceptual cohesion [19], code clones [6], co-location, and code authorship are promising. Our future research focuses on how to best combine such techniques.

Although we achieved good results using a weighted linear combination of execution trace and information retrieval scores [20], combining expert judgments is a core machine-learning problem. We feel that hybrid concern location techniques can further benefit from machine-learning solutions (e.g., “boosting” [14]) [23].

We cannot assume that CERBERUS or its constituent techniques will achieve a performance improvement in other contexts, or with other concern domains, programs, etc. As with any new technique, CERBERUS must be evaluated on more programs before we can be confident that our results are not context-specific.

The study data and source code for RHINO and CERBERUS are available online at http://www.cs.columbia.edu/~eaddy/concerntagger.

8. Acknowledgments

G. Antoniol was partially supported by the Natural Sciences and Engineering Research Council (NSERC) of Canada (Research Chair in Software Change and Evolution #950-202658) and by an NSERC Discovery Grant.

9. References

[1] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, E. Merlo, "Recovering Traceability Links between Code and Documentation," *IEEE Transactions on Software Engineering,* 28(10):970–983, 2002.

[2] G. Antoniol, Y.-G. Guéhéneuc, "Feature Identification: A Novel Approach and a Case Study," *Intl. Conf. on Software Maintenance (ICSM)*, 2006.

[3] R. Baeza-Yates, B. Ribeiro-Neto, *Modern Information Retrieval*: Addison-Wesley, 1999.

[4] T. J. Biggerstaff, B. G. Mitbander, D. Webster, "The concept assignment problem in program understanding," *Intl. Conf. on Software Engineering (ICSE)*, 1993.

[5] B. Boehm, "Software engineering," *IEEE Transactions on Computers,* C-25(12):1226–1241, 1976.

[6] M. Bruntink, A. v. Deursen, R. v. Engelen, T. Tourwe, "An evaluation of clone detection techniques for identifying cross-cutting concerns," *Intl. Conf. on Software Maintenance (ICSM)*, 2004.

[7] G. Canfora, L. Cerulo, M. D. Penta, "On the Use of Line Co-change for Identifying Crosscutting Concern Code," *Intl. Conf. on Software Maintenance (ICSM)*, 2006.

[8] K. Chen, V. Rajlich, "Case study of feature location using dependence graph," *Intl. Wkshp. on Program Comprehension (IWPC)*, 2000.

[9] M. Eaddy, A. Aho, G. C. Murphy, "Identifying, Assigning, and Quantifying Crosscutting Concerns," *Wkshp. on Assess. of Contemp. Modularization Techniques (ACoM)*, 2007.

[10] M. Eaddy, T. Zimmerman, K. D. Sherwood, V. Garg, G. C. Murphy, N. Nagappan, A. V. Aho, "Do Crosscutting Concerns Cause Defects?," *IEEE Transactions on Software Engineering (in press),* 2008.

[11] ECMA, "ECMAScript Standard," ECMA-262 v3, ISO/IEC 16262, 2007.

[12] T. Eisenbarth, R. Koschke, D. Simon, "Locating features in source code," *IEEE Trans. on Soft. Eng.,* 29:210–224, 2003.

[13] A. D. Eisenberg, K. De Volder, "Dynamic Feature Traces: Finding Features in Unfamiliar Code," *Intl. Conf. on Software Maintenance (ICSM)*, 2005.

[14] Y. Freund, R. E. Schapire, "A decision-theoretic generalization of on-line learning and an application to boosting," *Journal of Computer and System Sciences,* 55(11):119–139, 1997.

[15] J. H. Hayes, A. Dekhtyar, S. K. Sundaram, "Advanced Candidate Link Generation for Requirements Tracing: The Study of Methods," *IEEE Transactions on Software Engineering,* 32(1):4–19, 2006.

[16] E. Hill, L. Pollock, K. Vijay-Shanker, "Exploring the Neighborhood with Dora to Expedite Software Maintenance," *Automated Software Eng. (ASE)*, 2007.

[17] A. J. Ko, R. DeLine, G. Venolia, "Information Needs in Collocated Software Development Teams," *Intl. Conf. on Software Engineering (ICSE)*, 2007.

[18] D. Liu, A. Marcus, D. Poshyvanyk, V. Rajlich, "Feature Location via Information Retrieval based Filtering of a Single Scenario Execution Trace," *Automated Software Eng. (ASE)*, 2007.

[19] A. Marcus, D. Poshyvanyk, R. Ferenc, "Using the Conceptual Cohesion of Classes for Fault Prediction in Object Oriented Systems," *IEEE Transactions on Software Engineering (to appear),* 2008.

[20] D. Poshyvanyk, Y.-G. Guéhéneuc, A. Marcus, G. Antoniol, V. Rajlich, "Feature Location Using Probabilistic Ranking of Methods Based on Execution Scenarios and Information Retrieval," *IEEE Transactions on Software Engineering,* 33(6):420–432, 2007.

[21] M. P. Robillard, "Representing Concerns in Source Code," Thesis, CS Dept., University of British Columbia, 2003.

[22] M. P. Robillard, G. C. Murphy, "Concern Graphs: Finding and describing concerns using structural program dependencies," *Int'l Conf. Software Eng. (ICSE)*, 2002.

[23] D. Shepherd, J. Palm, L. Pollock, M. Chu-Carroll, "Timna: A Framework for Automatically Combining Aspect Mining Analyses," *Automated Software Engineering (ASE)*, 2005.

[24] D. Sheskin, *Handbook of parametric and nonparametric statistical procedures*, 4th ed.: Chapman & Hall, 2007.

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

[26] K. Wong, S. R. Tilley, H. A. Müller, M.-A. D. Storey, "Structural Redocumentation: A Case Study," *IEEE Software,* 12(1):46–54, 1995.

[27] W. E. Wong, J. R. Horgan, S. S. Gokhale, K. S. Trivedi, "Locating program features using execution slices," *App.-Specific Systems and Soft. Eng. (ASSET)*, 1999.

[28] W. Zhao, L. Zhang, Y. Liu, J. Luo, J. Sun, "Understanding How the Requirements Are Implemented in Source Code," *Asia-Pacific Soft. Eng. Conf. (APSEC)*, 2003.

[29] W. Zhao, L. Zhang, Y. Liu, J. Sun, F. Yang, "SNIAFL: Towards a Static Noninteractive Approach to Feature Location," *ACM Transactions on Soft. Eng. and Methodology,* 15(2):195–226, 2006.