Topology Analysis of Software Dependencies

MARTIN P. ROBILLARD
McGill University, Canada

Before performing a modification task, a developer usually has to investigate the source code of a system to understand how to carry out the task. Discovering the code relevant to a change task is costly because it is a human activity whose success depends on a large number of unpredictable factors, such as intuition and luck. Although studies have shown that effective developers tend to explore a program by following structural dependencies, no methodology is available to guide their navigation through the thousands of dependency paths found in a nontrivial program. We describe a technique to automatically propose and rank program elements that are potentially interesting to a developer investigating source code. Our technique is based on an analysis of the topology of structural dependencies in a program. It takes as input a set of program elements of interest to a developer and produces a fuzzy set describing other elements of potential interest. Empirical evaluation of our technique indicates that it can help developers quickly select program elements worthy of investigation while avoiding less interesting ones.

Categories and Subject Descriptors: D.2.6 [**Software Engineering**]: Programming Environments; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

General Terms: Algorithms, Experimentation, Human Factors

Additional Key Words and Phrases: Software evolution, software change, feature location, separation of concerns, static analysis, program understanding, software navigation

ACM Reference Format:

Robillard, M. P. 2008. Topology analysis of software dependencies. ACM Trans. Softw. Engin. Method. 17, 4, Article 18 (August 2008), 36 pages. DOI = 10.1145/13487689.13487691 http://doi.acm.org/10.1145/13487689.13487691

1. INTRODUCTION

Software projects typically go through multiple iterations during their lifetime [Kruchten 2000], with many iterations involving a number of modifications to

This article is a revised and extended version of a paper presented at ESEC/FSE 2005 in Lisbon, Portugal.

This research was supported by a McGill University start-up package, by the Natural Sciences and Engineering Research Council of Canada (NSERC), and by IBM.

Author's address: M. P. Robillard, School of Computer Science, McGill University, 3480 University Street, McConnell Engineering Building no. 318, Montreal, QC, Canada, H3A 2A7; email: martin@cs.mcgill.ca.

the source code. As part of most modification tasks, a developer must investigate the source code associated with the task prior to modifying it [Boehm 1976]. Ensuring that developers investigate source code efficiently is a challenging problem for software development organizations because it is an inherently human activity, whose success depends on a large number of unpredictable factors, such as intuition and luck.

In a previous empirical study of program investigation [Robillard et al. 2004], we observed that a distinctive characteristic of effective developers was their tendency to investigate source code by following structural dependencies. Although this practice can help make program investigation more systematic, it does not provide a complete strategy for software investigation. Indeed, in any nontrivial software system, the number of structural dependencies to follow is much too large to be completely covered by a developer. As a result, developers must rely on their intuition to determine where to look. In the case of expert developers working on a well-known system, intuition will generally do the trick. However, novice developers or developers working on an unfamiliar system may easily get stuck in irrelevant code and fail to notice important program functionality, leading to low-quality software modifications [Robillard et al. 2004].

This problem can be mitigated through a number of code searching tools and approaches (see Section 3). This paper provides a contribution to this corpus by investigating the hypothesis that the topology of structural program dependencies contains clues that can help identify elements that are likely to be more worthy of investigation than others. In other words, that patterns in the structural dependencies of a software system can indicate sections of code worthy of investigation, independently of the semantics of the program. Our motivation for investigating the potential of static dependency analysis is to develop a technique that can be used on incomplete or incorrect programs, and that is inexpensive enough to be used in a highly iterative fashion.

As part of our investigation, we developed an algorithm for a static analysis that identifies program elements likely to be of interest to a developer. Our algorithm takes as input a fuzzy set describing methods or fields of interest to a developer, and produces a fuzzy set containing methods and fields that are of potential interest. The degree of potential interest for each suggested element is obtained by analyzing two characteristics of the dependencies to elements in the set of interest: *specificity* and *reinforcement*. Informally, an element is specific if it is related to few other elements, whereas an element is reinforced if it is related to other elements of interest.

We implemented support for using our algorithm on Java systems in a tool called Suade. We report on two case studies and one experiment to evaluate the suggestions produced by our approach. Our results show that the algorithm produces suggestions that can significantly improve a developer's chances of identifying program elements associated with a task while avoiding code that is not relevant.

In the rest of this article, we first motivate the problem with a concrete scenario (Section 2) and present an overview of techniques previously proposed to help developers investigate source code (Section 3). We then present our algorithm (Section 4), and describe its current implementation for Java (Section 5).

We report on empirical evaluations of our technique in Sections 6 and 7, and conclude in Section 8.

2. MOTIVATION

We illustrate how our proposed technique can assist developers with a concrete scenario of software investigation. In this scenario, a developer is asked to modify the MARKERS feature of jEdit. jEdit is a programmer's text editor written in Java.[1] The MARKERS feature allows users to "bookmark" individual lines in a text file, and to navigate to marked lines though user-defined shortcut keys. In jEdit, the implementation of the MARKERS feature is far from trivial, and involves interactions between fields and methods scattered throughout at least 10 classes [Robillard 2006].

A developer unfamiliar with the code of jEdit might attempt to locate the implementation of the MARKERS feature through the time-tested strategy of searching the source code text for matches to a regular expression [Sillito et al. 2006]. Unfortunately, in our case, a search for the string "marker" through only the Java files of jEdit produces 517 matches. As a second attempt, let us assume that the developer proceeds with a more constraining query, and uses an advanced search tool that looks only for class declarations that begin with the string "marker." This strategy produces four results: classes `Marker`, `MarkerHighlight`, `MarkersMenuItem`, and `MarkersProvider`. After a brief inspection, the developer determines that all four of these classes indeed implement part of the MARKERS feature, and thus form a good starting point for investigating the rest of the code.

Where does the developer look next? The classes identified implement but a small piece of a complex puzzle. The developer needs to explore more of the code. But how to proceed? Our previous empirical work on software investigation provided evidence that effective developers tend to explore the source code by following structural dependencies [Robillard et al. 2004]. In our case, this would involve using a cross-reference tool to search for the parts of the program using or used by members of the four classes identified. Because our four starting-point classes define a total of 24 elements, searching for dependencies and managing the results promises to be difficult. Even with sophisticated tools capable of producing the desired results with a few simple queries, the developer still needs to inspect a list of results containing 68 entries,[2] with some results being very relevant (e.g., `Buffer.getMarkerInRange(int,int)`) and some, much less so (e.g., `VFSFileChooserDialog.VFSFileChooserDialog(...)`). By inspecting all of the results manually, the developer would eventually identify `getMarkerInRange` as relevant and proceed to discover that an important part of the MARKERS feature is implemented in the `Buffer` class.

The technique we present in this paper is intended to facilitate the task of inspecting a large number of software dependencies to identify the results worthy of detailed investigation, at any point of the modification task. Applying

[1]www.jedit.org. This scenario is based on the code of release 4.2-final.

[2]Corresponding to all the fields and methods that are directly related to the 24 initial elements through method calls or field accesses.

our technique to the case described above, a developer will obtain, in a single step taking just a few seconds, a list of the 68 elements related to the elements defined in the four starting-point classes. Furthermore, the list of results will be ordered, with elements determined the most likely to be relevant appearing at the top, and vice-versa. In this specific case, the top and bottom results are the two elements mentioned above (`Buffer.getMarkerInRange(int,int)`) and `VFSFileChooserDialog.VFSFileChooserDialog(...)`, respectively). Using this technique whenever required while exploring and modifying the code, developers should be able to spend more time understanding source code and less time examining long lists of search results. Indeed, anecdotal evidence collected by Sillito et al. documents how professional programmers quickly abandoned the inspection of search results "because the list was large and provided no way for the participants to differentiate the results (i.e., no information in the result list indicated what was relevant)" [Sillito et al. 2006, Section 5.2]. Our proposed technique directly addresses this issue by providing a ranking that can help developers focus on elements determined more likely to be relevant. Later sections of this paper will revisit the scenario described in this section to provide additional details about how the ranking is done, and why it works.

3. RELATED WORK

The goal of our topology analysis approach is to facilitate searching the source code of a software system at any point during investigation and modification tasks. As such, this technique presents an innovation in the line of code searching techniques typically proposed as part of integrated development environments (IDEs). However, our approach also can be used to complement a variety of approaches that have been proposed to help developers identify the source code related to high-level concepts. Such approaches usually come under the banner of *concept*, *concern*, or *feature location* approaches, and use a wide range of analysis techniques. We put our work in context by offering both a review of the historical development of code-searching techniques, as well as providing an overview of the feature location techniques they complement.

3.1 Code Searching Techniques

Support for code investigation and understanding was initially developed in the form of stand-alone lexical search tools (e.g., grep [Aho 1980]) and program databases (e.g., CIA [Chen et al. 1990], XREFDB [Lejter et al. 1992]). Basic program search and cross-referencing tools have also been provided as part of integrated development environments for many decades (e.g., in Interlisp [Sanella 1983], Smalltalk [Goldberg 1984], and Eclipse [Object Technology International, Inc. 2001]). More advanced techniques have also been proposed by the research community to experiment with different ways to exploit structural dependencies to help developers navigate source code. For example, the JQuery browser [Janzen and De Volder 2003] allows developers to browse and visualize software dependencies according to customized queries. Another potential way to reduce the effort and time associated with code navigation

is to perform searches in the background. The Mylar system generates automatic searches for dependencies of elements currently active in an integrated development environment [Kersten and Murphy 2005]. In the space of purely static-analysis-based code-searching techniques, the novelty of our research lies in the analysis of the *topology* of program dependencies, and its use to produce results ranked by degree of potential interest to a developer.

3.2 Program Slicing

Program slicing is a type of analysis intended to identify the parts of a program that may affect, or be affected by, the values computed at some point of interest [Tip 1995]. Slicing was originally defined as a static analysis technique [Weiser 1984], but dynamic variants have since been developed. For software evolution activities, slicing can be used to help determine the impact of changes [Gallagher and Lyle 1991]. Visual techniques have also been developed to help in this process [Gallagher 1996].

Although they are conceptually appealing techniques, static slicing and its variants suffer from practical limitations. First, computing slices can be expensive [Weiser 1984], and pragmatic considerations may require lower-precision data-flow analyses [Tonella et al. 1997]. Second, because a statement is often transitively dependent on many other statements, slices are often very large [Jackson and Rollins 1994; Weiser 1984]. Although techniques have been developed to help control the extent of the code covered by a slice [Orso et al. 2001], slicing remains a computationally expensive technique. In many code investigation situations, the fine-grained and precise results produced by slicing may not warrant the cost of computing and inspecting program slices.

3.3 Dynamic Analysis

A number of techniques can help developers navigate or reason about source code of interest using information collected as the program executes. A first category of such dynamic-analysis-based techniques includes techniques to automatically locate the code implementing a feature. For example, with the *Software Reconnaissance* [Wilde and Scully 1995] and *Execution Slices* [Wong et al. 1999] techniques, the code implementing a feature is determined by comparing a trace of the execution of a program in which a certain feature was activated to one where the feature was not activated. Another approach to feature location based on dynamic analysis was developed by Eisenbarth et al. [2003]. Eisenbarth et al. produce the mapping between components and test cases using mathematical concept analysis (a partial ordering and clustering technique [Snelting 1998]). In addition to producing a basic mapping between components and test cases, the approach of Eisenbarth et al. involves the refinement of the feature-to-code mapping through inspection by a developer of a static dependency graph of the analyzed program. This step helps achieve a more precise and complete description of the code that implements a feature, but requires additional effort on the part of the developers. A number of other techniques have also been proposed that address the same problem while providing additional functionalities, such as a probabilistic ranking of the results

or tools to analyze the interaction of features [Antoniol and Guéhéneuc 2005; Eisenberg and De Volder 2005; Salah and Mancoridis 2004].

Other categories of dynamic-analysis-based approaches include dynamic slicing and impact analysis. Dynamic slicing [Agrawal and Horgan 1990; Gyimóthy et al. 1999] is a variant of slicing that takes into account program execution trace information. Specifically, dynamic slicing only considers program dependencies that occur in a specific execution of the program. Dynamic impact analysis techniques [Orso et al. 2004] help developers reason about the sections of the code (typically methods or functions) that can be impacted by modifications to other areas of the code.

In contrast to static approaches, dynamic approaches depend on the availability and quality of test cases for an executable system. As such, they cannot be applied to incomplete code or to code that cannot be executed. In the specific case of dynamic feature location, the techniques can only identify the code relevant to features that can be expressed at the user level. These form a proper subset of the concerns a developer might wish or need to investigate. Often, developers must investigate code overlapping different features to understand enough of the system to respect the existing design. Because it is independent of the execution of specific features, our static approach does not suffer from this limitation.

3.4 Information Retrieval

Another approach taken to identify the code associated with a feature is to use information retrieval techniques. Antoniol et al. [1999] proposed an approach to determine a set of components potentially affected by a maintenance task using a probabilistic analysis of the text of the maintenance request. This approach, however, produces results only at the granularity of high-level components (classes), and cannot be used to identify more fine-grained elements such as methods.

Marcus et al. [2004] propose to use an information retrieval technique called Latent Semantic Analysis (LSI) to automatically map concepts expressed in natural language to the corresponding source code in a software system. In this interactive approach, users write or select queries which are then evaluated to return sections of code (e.g., functions) whose text is similar to that of the query.

The SNIAFL technique of Zhao et al. [2004] combines an analysis of the names of functions and identifiers with a call graph analysis to automatically identify the functions associated with a textual description of a feature. The main tradeoff of SNIAFL is that a developer must produce a description of *all* features in a system in order to be able to fully use the technique.

3.5 Repository Mining

A number of approaches can help developers identify elements of interest in the context of a software modification task through analysis of a repository of software artifacts. Both Zimmermann et al. [2004] and Ying et al. [2004] proposed data mining techniques that report on elements that are often changed together during program evolution tasks. This information can help a developer

determine where to look when investigating source code. The advantage of data mining approaches is that, given enough evidence, the elements recommended have the potential to be highly relevant. The main drawback of these approaches is the necessity of having a large history of changes available for analysis. This requirement is especially true if results are to be computed and reported at the level of class members. Reliance on change history implies that the approach cannot be used when tasks address code that was never changed before.

4. ALGORITHM

In developing the advanced code searching technique we present in this paper, our overarching goal was to help developers investigate source code in an interactive way, and with as little ancillary effort as possible (i.e., without the requirement of writing queries, tests cases, feature descriptions, etc.).

4.1 Hypothesis and Heuristics

To meet our goal, we were interested in investigating the hypothesis that the topology of structural program dependencies contains clues that can help identify elements that are likely to be more worthy of investigation than others. We derived this hypothesis after analyzing in detail how developers investigate source code, as part of a previous empirical study of programmers [Robillard et al. 2004]. Analyzing the topology of program dependencies is well-suited to our design goals as all the information required is readily available in the source code of the program under investigation.

We have thus developed an algorithm for suggesting elements to examine during a program investigation task based on the topology of structural dependencies in a program. The algorithm takes as input a *set of interest* $\bar{I}$. This set contains program elements (e.g., fields and methods) identified by a developer as interesting in the context of the task.[3] Our algorithm then analyzes the structural dependencies between the elements in $\bar{I}$ and the rest of the program, and produces a *suggestion set* $\bar{S}$ containing elements related to $\bar{I}$ with, for each element in $\bar{S}$, a value indicating its potential interest to the developer. In brief, our algorithm searches for all the *structural neighbors* of a set of program elements, and returns these elements, ranked in order of potential interest for the developer. The critical aspect of our algorithm is the method we use to produce the ranking. This ranking is based on a set of heuristics derived from two relatively simple intuitions: *specificity*, and *reinforcement*.

Specificity. According to the intuition of specificity, structural neighbors that have few structural dependencies are more likely to be interesting because their relation to an element of interest is more unique. Figure 1 illustrates this concept with two examples. In this figure, elements in gray represent elements of interest as flagged by a developer. In example *a*), element of interest `A` is referenced (e.g., called) by five other elements, whereas element `B` is referenced by a single element. In this scenario, the set of interest consists of two elements (`A` and `B`) and the set of structural neighbors contains six elements (elements

[3]In the context of our algorithm, we take "interesting" to mean "worthy of detailed investigation".

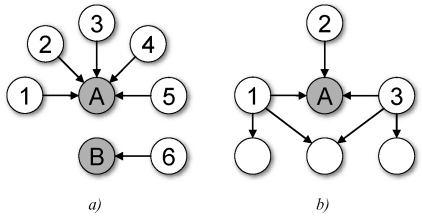

Fig. 1. Sample program graphs to illustrate specificity.

1 through 6). Because element B is related to only a single other element (6), we say that element 6 is more *specific* to the set of interest. In a ranking of structural neighbors, element 6 would be ranked higher based on specificity.

Example *b*) shows how specificity can also be taken into account for the relation that is the *transpose* of the one used to derive structural neighbors. In this case, element of interest A is referenced by three other elements. Elements 1 and 3 refer to two elements besides A. However, element 2 refers to *only* A, and to no other element. We would thus consider element 2 to be more specific, and rank it as more interesting with respect to the other structural neighbors.

The intuition of specificity is based on the reasoning that elements that are very specific to a set of interest probably contribute to the implementation of the concept, feature, or concern associated with the set of interest. For example, in Figure 1 *a*), element A is more likely to be a generic service method because it has a higher fan-in, and one (potentially) stands to learn less from inspecting its callers as from the caller of B, which could be a method that is on a call path that is strongly associated with the implementation of the feature under scrutiny.

In the scenario presented in Section 2, the top element (getMarkerInRange) is very specific to the set of interest because it calls a single method, and this method is in the set of interest. In contrast, the structural element with the lowest degree of interest (the constructor of VSFFileChooserDialog) is a structural neighbor because it calls a method in the set of interest that is called by 80 other methods. Hence, this element is not very specific.

Reinforcement. According to the intuition of reinforcement, structural neighbors that are part of a cluster that contains many elements *already in the set of interest* are more likely to be interesting because they are the "odd ones out" in the cluster of elements related to the set of interest. By cluster, we mean a group of elements found by following a specific type of dependency from a single element (e.g., a cluster formed by all the callers of method m). Figure 2 illustrates this concept with two examples. As in Figure 1, elements in gray represent elements of interest as flagged by a developer. In example *a*), element of interest A is referenced (e.g., called) by five elements, none of which are part of the set of interest. As such, we do not consider the elements 1 to 5 to be reinforced. In contrast, element B is also referenced by five other elements, all of which are flagged as interesting except for element 6. For this reason, we say that 6 is heavily reinforced and likely to be interesting.

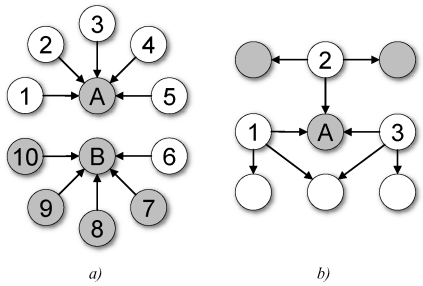

Fig. 2. Sample program graphs to illustrate reinforcement.

Example *b*) shows how reinforcement can also be taken into account for the relation that is the *transpose* of the one used to derive structural neighbors. In this case, element of interest `A` is referenced by three other elements, all of which refer to two elements in addition to `A`. Elements `1` and `3` refer to two elements that are not flagged as interesting. However, element `2` refers to *only* elements that have been flagged as interesting. For this reason, element `2` is considered reinforced, and would be ranked as more interesting with respect to elements `1` and `3`.

The intuition of reinforcement is based on the reasoning that elements that are reinforced probably share some structural property that associates them to the code of interest, and so it may be desirable (and the developer may be intending) to investigate all of the elements with that property.

In the concrete scenario presented in Section 2, the top element (`getMarkerInRange`) is heavily reinforced because all of its callers *and* all of its callees are in the set of interest.

Discussion. The use of specificity and reinforcement to determine the degree of interest of the structural neighbors of a set of interest is, of course, heuristic. For this reason, in some software investigation situations, specific or reinforced elements will not correspond necessarily to the more interesting elements among the structural neighbors. However, we are working with the hypothesis that specificity and reinforcement are indicative of potential interest in general. The evaluation of our approach (Sections 6 and 7) will provide supporting evidence for this hypothesis.

In the remainder of this section, we present an algorithm to calculate a metric representing the degree of interest of a structural neighbor of an element of interest based on our specificity/reinforcement criterion. In designing this algorithm, our guiding principle was to develop a representation of specificity and reinforcement that would be as simple as possible. There are two main reasons for this choice. First, a simple algorithm facilitates a baseline evaluation of the approach, which should prove easier to evaluate and improve incrementally than a sophisticated scheme. Second, because the results of the approach must appear sensible to the users, we strove for a relatively straightforward procedure that could be performed manually, at least on simple examples.

4.2 Basic Definitions

Because program investigation is an imperfect process, and because the results produced by our algorithm indicate a *degree* of potential interest, we use fuzzy sets to represent both the set of interest (input) and the suggestion set (output). The human-centric nature of program investigation makes fuzzy logic a particularly well-suited theory supporting our algorithm. In our application of fuzzy set theory, we use notation and definitions consistent with the presentation of Zimmermann [1996]. In particular, set variables with an overbar distinguish fuzzy sets from normal (crisp) sets.

Our algorithm relies on the concepts defined below. These concepts assume the existence of a program P to which the algorithm is applied. Formally, $P = (E, R)$ consists of a set of elements E and a set R of relations among these elements.

Definition 4.1 (*Program Element*). A program element $e \in E$ is any element that can be individually investigated by a developer.

Typical program elements in an object-oriented language include fields and methods. Although classes can fit the definition, in practice the amount of code forming their declaration is too large for them to constitute a unit of investigation for the purpose of our algorithm.

Definition 4.2 (*Relation*). A relation $r = (l, e_1, e_2) \in R$ is a program dependency of type l between two program elements e_1 and e_2.

Typical relations in an object-oriented language include field accesses and method calls.

Definition 4.3 (*Transpose*). Given a relation $r = (l, e_1, e_2) \in R$, its transpose is defined as $r^\top = (l^\top, e_2, e_1)$. In any program, all relations have a transpose, that is, $r \in R \Longrightarrow r^\top \in R$.

For example, if e_1 calls e_2, then e_2 is called by e_1.

Definition 4.4 (*Set of Interest*). Given a program $P = (E, R)$, a set of interest $\bar{I} = \{(e, \mu_{\bar{I}}(e)) \mid e \in E, \mu_{\bar{I}} : E \rightarrow [0, 1]\}$ is defined as a fuzzy set with membership function $\mu_{\bar{I}}$.

Definition 4.5 (*Suggestion Set*). Given a program $P = (E, R)$, a suggestion set $\bar{S} = \{(e, \mu_{\bar{S}}(e)) \mid e \in E, \mu_{\bar{S}} : E \rightarrow [0, 1]\}$ is defined as a fuzzy set with membership function $\mu_{\bar{S}}$.

In practice, the normalized membership functions $\mu_{\bar{I}}$ and $\mu_{\bar{S}}$ are specified as sets of ordered pairs, where the first element denotes a program element and the second its degree of membership [Zimmermann 1996]. For example: $\mu_{\bar{I}} = \{(e_1, 0.5), (e_2, 0.7)\}$.

In our proposed approach, the set of interest is a fuzzy set to allow the application of the algorithm on sets whose elements are not all associated equally strongly with the concept represented by the set. This feature supports the iterative application of our approach (in which the fuzzy suggestion set can become the set of interest), but also to use the approach with other techniques that

1: **Param:** $\bar{I}$: Set of interest
2: **Param:** L: Set of relation types to analyze
3: **Param:** $0 \leq \alpha \leq 1$: A calibration parameter
4: **Var:** $\bar{S} = \{\}$: Suggestion set
5: **Var:** $\bar{T} = \{\}$: Temporary set
6: **for all** $l \in L$ **do**
7: $\quad \bar{T} = analyzeRelation(l, \bar{I}, \alpha)$
8: $\quad \bar{S} = \bar{S} \uplus \bar{T}$
9: **end for**
10: **return** $\bar{S}$

Fig. 3. Main algorithm.

produce fuzzy sets of interest (e.g., Dynamic Feature Traces [Eisenberg and De Volder 2005]). We detail how we take into account the fuzziness of the input set in Section 4.4.

4.3 Main Algorithm

Figure 3 presents our high-level analysis algorithm. This algorithm is designed to be able to handle a structural dependency graph created from an open-ended set of relation types.

For each relation type considered (line 6), we calculate a temporary fuzzy suggestion set based on the relation type (line 7).[4] For example, using $l =$ *calls* will generate a temporary suggestion set based on the analysis of the methods called by methods in $\bar{I}$. At this point the temporary set corresponds to the structural neighbors of $\bar{I}$ according to relation l, and ranked by degree of interest according to the topology of the graph corresponding to l and its transpose. After a temporary suggestion set is calculated for a relation type, this set is merged with the final result set $\bar{S}$ (line 8).

Instead of merging the fuzzy sets obtained for each relation using the standard union operator for fuzzy sets,[5] we define a new operator $\uplus$ that works slightly differently: if an element x is in the intersection of two fuzzy sets, the resulting membership degree is *higher* than both maximums, and calculated using the following function:

$$\mu_{\bar{S}_1 \uplus \bar{S}_2}(x) = \mu_{\bar{S}_1}(x) + \mu_{\bar{S}_2}(x) - \mu_{\bar{S}_1}(x)\mu_{\bar{S}_2}(x). \quad (1)$$

We designed our union operator (Equation (1)) to be commutative, associative, and, for domain values in [0,1], to have a range in [0,1] that is always greater than (or equal to) the maximum of its operands. This last property is intended to reflect the intuition that if an element is found in the sets generated through different relations, these repeated occurrences reinforce each other. For example, according to this function, an element $(x, 0.75) \in \bar{S}_1$ intersecting with an element $(x, 0.50) \in \bar{S}_2$ will result in an element with membership 0.875 in the merged set.

[4]The set of relation types is a crisp set (no overbar).
[5]For two fuzzy sets $\bar{S}_1$ and $\bar{S}_2$ with membership functions $\mu_{S_1}(x)$ and $\mu_{S_2}(x)$ we usually have $\mu_{S_1 \cup S_2}(x) = \max(\mu_{\bar{S}_1}(x), \mu_{\bar{S}_2}(x))$.

The goal of our redefined union operator is to assign more relevance to elements found through multiple relations. For example, a method that both accesses a field of interest *and* calls a method of interest will have a higher degree value with our redefined union operator than if we had used the standard operator. In all cases where an element is found through a single relation (i.e., the intersection between all the $\bar{T}$ sets is the empty set), there will be no difference in the value produced. The impact of Equation (1) is that elements that are connected through multiple relations will tend to rise to the top of the suggestion list. So far our experience with the approach using a small number of relation types has not invalidated this strategy. However, it is important to note that Equation (1) introduces a dependency between the results of the approach and the nature of the set of relations R. In particular, using our redefined merge operator with a graph comprised of a large number of heavily correlated relations might be problematic. In the improbable case where analyzing such a graph would be required, it might be worthwhile to consider reverting to the traditional definition of the fuzzy union operator.

4.4 Analyzing Relations

The key step of the approach is to produce the membership function for the suggestion set. This step is abstracted as the *analyzeRelation* function in Figure 3. In essence, this function produces a degree value for all of the structural neighbors of elements in $\bar{I}$ in the simplest way possible that takes into account both specificity and reinforcement, and both the direct and transpose (inverse) relations. These four cases correspond to the four cases represented in Figures 1 and 2. We note again that we designed our algorithm to perform calculations on relations and their transpose because both directions of a relation can provide clues that an element might be worthy of investigation. Section 4.1 describes such potential scenarios.

We define the function *analyzeRelation* in Figure 4 and illustrate the algorithm with the graph of Figure 5. For simplicity, we can consider that Figure 5 is a call graph, with elements representing methods and arrows representing a method call. Methods `A` and `B` have already been flagged as methods of interest. In our example, we compute *analyzeRelation* for the *called by* relation.[6]

$$degree = \left(\frac{1 + |S_{\text{forward}} \cap \bar{I}|}{|S_{\text{forward}}|} \cdot \frac{|S_{\text{backward}} \cap \bar{I}|}{|S_{\text{backward}}|} \right)^{\alpha}. \tag{2}$$

For each element x in a set of interest (line 8), this function obtains the range of relation l corresponding to domain x and stores it in a temporary set S_{forward} of elements (line 9). Let us perform the calculation for `A`. Line 9 yields $S_{\text{forward}} = \{c, d, e\}$.

Then, each range element $s \in S_{\text{backward}}$ that is not already in the set of interest (lines 10–11) is added to the suggestion set. In our case we add `c`, `d`, and `e` to the suggestion set. The degree value for each element is calculated in three steps. First, we obtain the set of elements S_{backward} (line 12). In the

[6]The direction of the *called by* relation is thus opposed to the direction of the arrows in the call graph represented by Figure 5.

1: **Assumes:** $P = (E, R)$: A program
2: **Param:** $\bar{I} = \{(x, \mu_{\bar{I}}(x)) \mid x \in E\}$: Set of interest
3: **Param:** $l \in \{r \mid (r, e_1, e_2) \in R\}$: Relation type to analyze
4: **Param:** $0 \leq \alpha \leq 1$: A calibration parameter
5: **Var:** $S_{\text{forward}} \in E$: A (crisp) set of program elements
6: **Var:** $S_{\text{backward}} \in E$: A (crisp) set of program elements
7: **Var:** $\bar{Z}$: The (fuzzy) set to be returned
8: **for all** $x \in \bar{I}$ **do**
9: $S_{\text{forward}} = \{y \mid (l, x, y) \in R\}$
10: **for all** $s \in S_{\text{forward}}$ **do**
11: **if** $s \notin \bar{I}$ **then**
12: $S_{\text{backward}} = \{y \mid (l^{\top}, s, y) \in R\}$
13: See Equation 2
14: $\bar{Z} = \bar{Z} \cup \{(s, (\mu_{\bar{I}}(x) \cdot \text{degree})\}$
15: **end if**
16: **end for**
17: **end for**
18: **return** $\bar{Z}$

Fig. 4. Function *analyzeRelation*.

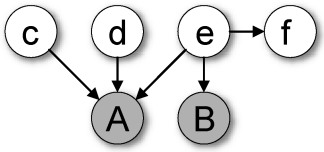

Fig. 5. Example program graph.

remainder of our example, we will calculate the degree value for $s =$ e. In the case of e, $S_{\text{backward}} = \{A, B, f\}$. Second, we calculate the degree value for e using Equation (2) (line 13). Presuming $\alpha = 0.25$ (for now), we have:

$$\textit{degree} = \left(\frac{1+0}{3} \cdot \frac{2}{3}\right)^{0.25} = 0.69.$$

Third and finally, we add the new element to $\bar{Z}$, the suggestion set to be returned.[7]

The design of Equation (2) can be justified as follows. This equation calculates a degree value for a structural neighbor by evaluating the specificity and reinforcement for both forward and backward references. The two terms inside the parentheses correspond to a computation of the degree for the forward (left term) and backward (right term), respectively. For a given term, the numerator measures the strength of the reinforcement, and the denominator measures the strength of the (inverse) specificity. The additional unit in the numerator of the forward term accounts for the cases where there is no reinforcement. This term is not necessary for the backward term as the set S_{backward} will always include at least the one element used to generate S_{forward} (s in Figure 4). Finally, the value obtained by multiplying both terms is taken to an exponent

[7]The union operation of line 13 uses the traditional definition of unions for fuzzy sets (using the maximum values of membership functions).

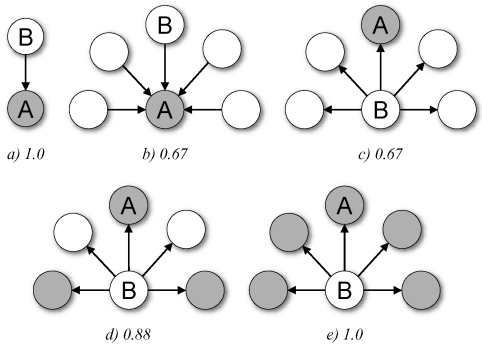

Fig. 6. Degree calculations for sample patterns.

α in the unit interval. This last treatment has a relatively minor impact, and is only present to allow tool implementors to scale all degree values up to obtain a more uniform distribution.

As can be seen from Equation (2), reinforcement is orthogonal to specificity and can potentially compensate for it. In other words, an element related to many other elements that would be ranked very low according to the specificity criterion could end up being ranked highly if all the elements it is related to have been flagged as interesting.

4.5 Complexity

The space complexity of our suggestion algorithm is negligible as it only needs temporary storage for small subsets of an entire program.[8] The time complexity is linear in the cardinality of the set of interest $\bar{I}$ used as input to the algorithm. More precisely, given the inputs L (set of relations) and $\bar{I}$ (set of interest), and assuming that the upper bound on the number of dependencies of a program element is a small constant, the execution time of the algorithm can be modeled as $O(|L| \times |\bar{I}|)$.

4.6 Example Applications

We present a number of sample degree value calculations, the study of which can provide additional insights into the behavior of the algorithm. Figure 6 shows five example topologies. In each example, `A` is a element in $\bar{I}$ and `B` an element in S_{forward}. Elements in grey are in the set of interest $\bar{I}$. For each pattern, we include the degree value for `B` as calculated with Equation (2), with $\alpha = 0.25$. For consistency with the call graph example of Section 4.4, the arrows in Figure 6 are oriented in the direction of the transpose of the relation used to perform the calculation.

Example *a)* shows the case where `A` is referred to by a single element that does not refer to anything else. In this situation our algorithm returns a degree of 1.0. This is the highest possible quantification of potential interest, and adequately represents the fact that `A` has (apparently) no purpose besides contributing to the implementation of `B`.

[8]The static analysis required to execute the algorithm is discussed in Section 5.2.

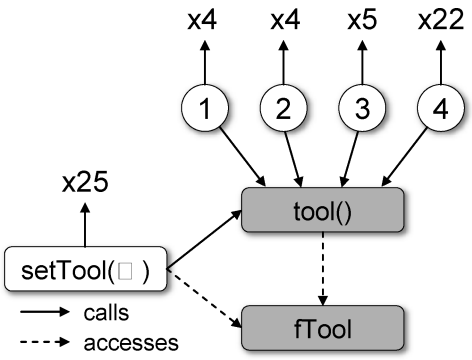

Fig. 7. Partial dependencies in JHotDraw.

Examples *b*) and *c*) represent cases where suggested elements are not reinforced. In the case of *b*), we have a specificity of 1/5 for the direct relation and a specificity of 1/1 for the transpose, yielding $(1/5)^{0.25} = 0.67$. Case *c*) is symmetrical to *b*), but in this case the specificity for the direct relation is 1/1 and the specificity for the transpose relation is 1/5, yielding an identical value of 0.67.

Example *d*) has the same configuration as *c*), but in this case `B` is reinforced because two of the elements in S_{backward} are in $\bar{I}$, yielding the term 1/1 for the direct relation and 3/5 for the transpose, for a final value of $(3/5)^{0.25} = 0.88$.

Finally, *e*) represents another extreme case, where `A` is referred to by a single element `B`, and everything that `B` refers to is already in $\bar{I}$. In this case we also obtain the maximum degree of 1.0 as it is not possible to be more reinforced.

As these examples illustrate, a degree value in isolation is much less useful that a degree value as interpreted relatively to the degree value of other suggestions. Hence, in a large group of structural neighbors, elements involved in a topology similar to that in *a*), *d*), or *e*) would be ranked as more potentially interesting than elements involved in topologies with weaker combinations of specificity and reinforcement.

We conclude this series of examples with a concrete scenario taken from the JHotDraw drawing application (version 5.3).[9]

As a set of interest, we choose two elements in class `DrawApplication`: method `tool()` and field `fTool`, and apply the algorithm with the relations *called by*, *calls*, *accesses*, and *accessed by* and parameter $\alpha = 0.25$. In this case we chose a set of interest with a field and a method related to the concept of "tool management" to show how the algorithm operates with multiple types of relations (i.e., field access and method call).

We start with the execution of *analyzeRelation* with $l = \textit{called by}$. Obtaining S_{forward} for `tool()` (line 9 of Figure 4) yields five callers, represented in Figure 7 by the four numbered nodes and the `setTool(...)` node. In the figure, elements in the set of interest $\bar{I}$ are shaded gray. Because none of the five elements is in $\bar{I}$, their direct degree is 1/5 (Equation (2)). For each of the five methods in

[9]http://www.jhotdraw.org

S_{forward}, the range of the transpose relation (*calls*) is calculated, and shown on the figure by an arrow indicating the number of callees. For example, method `setTool(...)` calls 26 methods: `tool()` and 25 other ones (names not important). Because `fTool` is a field and not called by any method, a first suggestion set (line 6 of Figure 3) can be generated for the *called by* relation:

method	1	2	3	4	`setTool(...)`
degree	0.45	0.45	0.43	0.31	0.30

At this point the degree for `setTool(...)` as calculated using the *called by* relation is relatively low. According to Equation (2), this value is calculated by multiplying 1/5 (direct relation) by 1/26 (transpose relation) and taking the result to the power of 0.25. The value is low mostly because `setTool(...)` has low transpose specificity with respect to the *called by* relation (it calls 25 methods besides `tool()`).

The second iteration analyzes the relation *calls* and yields an empty suggestion set because none of the elements in the set of interest call anything. The merge operation thus produces exactly the suggestion set above.

The third iteration analyzes the relation *accesses*. This analysis also yields an empty suggestion set because `tool()` only accesses the field `fTool` and this element is already in the set of interest.

The final iteration analyzes the relation *accessed by* and considerably changes the suggestion set. The method `tool()` is of course not accessed by anything, so the analysis focuses on `fTool`. As shown in Figure 7, `fTool` is accessed by only two methods (`tool` and `setTool`), yielding a high specificity. Furthermore, one of the methods (`tool()`) is already in the set of interest, yielding a high reinforcement for the remaining range (method `setTool(...)`). The degree for the direct relation is thus $(1 + 1)/2 = 1$. In addition, `setTool` only accesses a single field, `fTool` itself. This yields a transpose degree of $1/1 = 1$. As a result, the final degree for `setTool(...)` is 1. Finally, using our redefined fuzzy union operator (Equation 1), we merge the previously calculated suggestion set (shown above) with the set {(`setTool`(...), 1.00)} (as calculated for the relation *accessed by*). The final suggestion set becomes:

method	1	2	3	4	`setTool(...)`
degree	0.45	0.45	0.43	0.31	1.00

This result has a meaningful application because `setTool` is a nontrivial mutator of `fTool`, and would likely need to be investigated by a developer interested in understanding the mechanism for managing drawing tools in JHotDraw. In this simple case, the name of the method is a good indication of its relevance to the set of interest. However, in class `DrawApplication`, a total of six elements besides `tool()` and `fTool` have an identifier that contains the string "tool", yet only `setTool` is directly structurally related. In addition, in the case where large numbers of dependencies must be considered, developers may not always deem it cost-effective to read the name of each element returned in the result of a

cross-reference search. In such cases, our technique can help by automatically ranking elements based on our specificity–reinforcement criterion.

5. CURRENT IMPLEMENTATION

We implemented an instance of our topology analysis algorithm to analyze Java programs using the four relations: *calls*, *called by*, *accesses*, and *accessed by*. We developed this prototype as a plug-in for the Eclipse platform.[10] Eclipse is an integrated development environment with an architecture that supports the addition of modules, called *plug-ins*, that add to the environment's functionality. The standard distribution of Eclipse includes a set of plug-ins that provide extensive support for development in the Java programming language. Our plug-in, called Suade [Weigand Warr and Robillard 2007], lets the user specify a set of interest, apply the algorithm to the set, and view the results of the analysis. Suade is freely available for download from http://www.cs.mcgill.ca/~swevo/suade.

5.1 Using the Suade Plug-In

A user specifies the set of interest using the ConcernMapper plug-in [Robillard and Weigand Warr 2005]. ConcernMapper is a separate tool developed and deployed independently from Suade, but that nevertheless forms an integral part of the tool support for our topology analysis algorithm. The ConcernMapper plug-in augments Eclipse with a view in which a user can specify different *concerns* and add different program elements (such as fields and methods of Java classes) to each concern by dragging elements from other Eclipse views such as the Package Explorer or the Search Results view.

Figure 8 shows a view of Eclipse during a usage scenario involving ConcernMapper. The integrated development environment consists of four main views, each forming a quadrant. The Package Explorer is in the upper left quadrant. This standard Eclipse view shows the declarative structure of a Java project and allows users to browse the elements of a project (packages, compilation units, classes, methods, fields, etc.) and to select elements to display in an editor. The upper-right quadrant displays editors for the resources (files) in a project. The Search Results view is in the lower-right quadrant. This standard Eclipse view shows the results of cross-reference or lexical searches that can be performed through the features of the environment (e.g., to obtain all the methods accessing a specific field). Finally, the ConcernMapper view is in the lower-left quadrant. This view allows users to associate various program elements with a concern name (the name of a high-level concept of interest in the context of a software development or evolution task). A user creates a new concern by clicking on a button in the view's tool bar and by entering a name for the concern (e.g., "Storage"). Then, the user associates elements in a project with the concern by dragging elements from any Eclipse view and dropping them on the box representing the concern. When elements are added to the concern, they become highlighted in bold in the other Eclipse views and the name of the concern

[10] www.eclipse.org

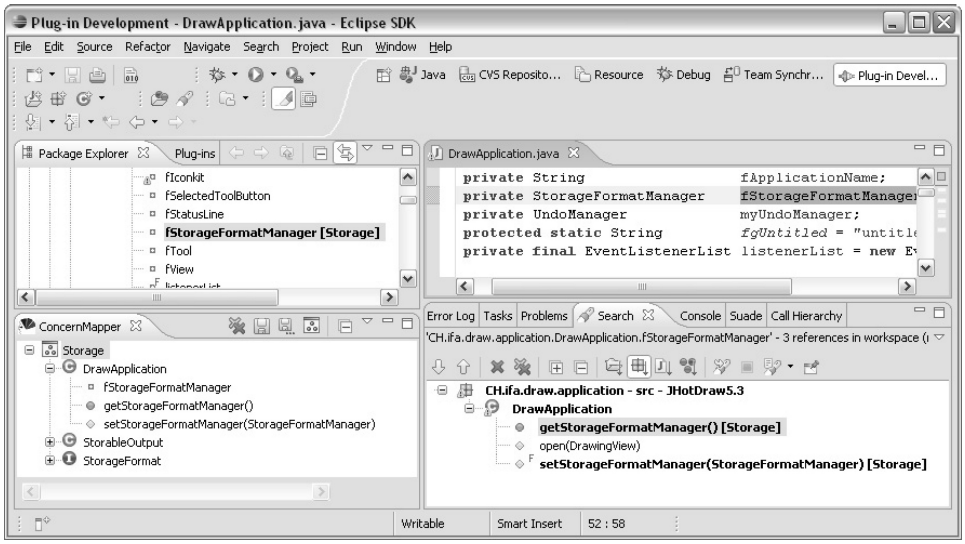

Fig. 8. The ConcernMapper Eclipse plug-in.

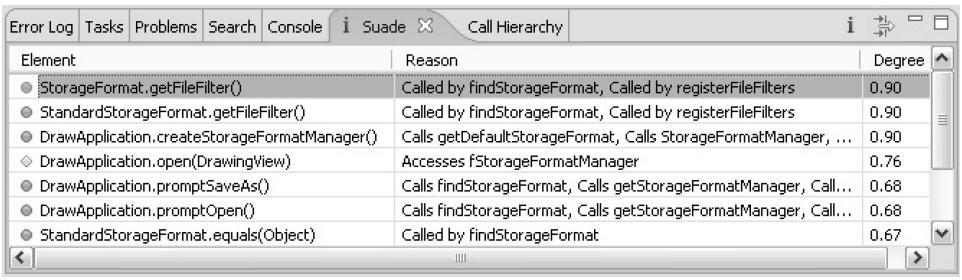

Fig. 9. The Suade view.

is appended to their name. In our example, a developer is interested in the implementation of the permanent storage of drawings in a drawing program. The user created a "Storage" concern in ConcernMapper and added elements from the classes `DrawApplication`, `StorableOutput`, and `StorageFormat`. In Figure 8, the Package Explorer and Search Results views show how these elements are highlighted in other views. In general, ConcernMapper supports change tasks by allowing developers to group all the elements related to a task in a single view, and to save this information for later use. Users can also specify a degree of relevance to the concern for any element by using the slider at the bottom of the view. In the context of the Suade plug-in, the elements in the ConcernMapper view form the set of interest used as input to the algorithm.

To apply the algorithm to a set of interest, a user must open the Suade view (Figure 9) and drag the concern representing the set of interest into the view. This will trigger a computation of the algorithm (using a fixed value of $\alpha = 0.25$). Once the algorithm completes, the view is refreshed with the results of the computation.

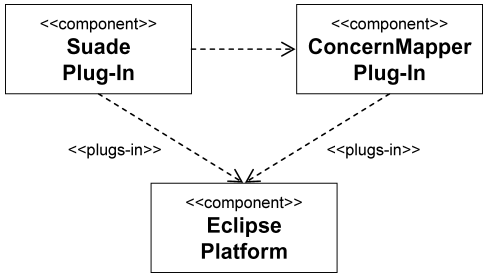

Fig. 10. Architecture of the Suade plug-in.

Figure 9 shows the details of the Suade View with an example of results. This view contains a table listing, on each row, one element of the suggestion set. The leftmost column (Element) contains a textual identifier and an icon for the element. The middle column (Reason) provides additional information about how the element was found. This aspect of the results is not formally specified by the algorithm; In our case we simply list each relation that took part in the calculation. Finally, the last column (Degree) shows the membership degree for the suggested element. For example, three elements from three different classes are recommended with a degree of 0.90. Because it is expected that suggested elements will often be of interest, the Suade View includes functionality to display the code of a suggested element and to add a suggested element to the set of interest in the ConcernMapper view.

5.2 Architecture and Implementation

Figure 10 presents the architecture supporting the deployment of the Suade plug-in. This architecture comprises three main components: the Eclipse platform (including all of its standard plug-ins), the ConcernMapper plug-in, and the Suade plug-in and its dependencies. Although Suade is deployed separately from ConcernMapper, it cannot function independently.

The responsibility of ConcernMapper in this implementation is to manage an internal model representing the set of interest. This model is exported through a simple Façade [Gamma et al. 1995] application programming interface (API) allowing other Eclipse plug-ins to access this model. When a user requests suggestions (by dragging a concern from the ConcernMapper View to the Suade View), Suade queries the concern model to get a list of elements of interest, applies the suggestion algorithm by analyzing the source code of the project on which the set of interest is defined, and displays the results in the Suade view.

The static analysis necessary to obtain the different relations necessary for the algorithm is performed using an specially-tailored version of the JayFX fact extractor.[11] JayFX parses the source code of a Java project and builds an in-memory database representing the structural dependencies among the elements in the project. Currently, we use JayFX to extract method calls and field accesses (and their transpose). The semantics of the “calls” relation includes calls between a caller and (potentially) dynamically-bound method

[11]http://www.cs.mcgill.ca/~swevo/jayfx.

implementations. JayFX infers these relations by performing a class hierarchy analysis (CHA).

5.3 Design Decisions

In developing the technology supporting our topology analysis of dependencies, we have experimented with a number of design alternative through a sequence of unreleased prototypes. The first public release of Suade is thus the result of development efforts that have benefited from over 18 months of experimentation. This experimentation has led to a number of important design and implementation decisions.

Fixing the Alpha Value. One of the explicit design dimension for our implementation was the value of the α parameter used to scale the degree value of suggested elements (see Figure 4, line 4). The purpose of this parameter is simply to scale up the degree values, because the application of Equation (2) without using α results in very small degree values. For example, without taking into account the influence of α, the degree values for the callers of `tool()` in the scenario of Figure 7 are 0.040, 0.033, 0.0087, and 0.0077, respectively. Such small values require more precision to represent, and are potentially more difficult to interpret. Using a scaling factor such as α allows us to have a more uniform spread of values. Theoretically, for a single relation, different values of alpha do not modify the order among suggested elements. The only potential consequence of α on the order of suggested elements is caused by its impact on the results of the fuzzy union operator for suggestion sets (Equation (1)). However, given the heuristic nature of the approach, we hypothesize that the impact of any specific α value on the overall usability of the approach is negligible. We determined a concrete value for alpha by conducting an experiment in which we generated a large number of suggestion sets and studied the distribution of degree values for different values of α [Robillard 2005]. This experiment indicated that a value of 0.25 produced a uniform spread of degree values. We also experimented with different α values during the development our our prototypes and found that 0.25 indeed produced a completely satisfactory distribution. This value has been chosen for Suade release 0.0.1. Although experimentation with our current prototype has shown $\alpha = 0.25$ to be adequate, use of the technique on dependency graphs involving different relations or for different programming languages might warrant adjustments.

Choosing the Set of Relations. Another explicit design dimension for our approach is the set of relation types to use for generating suggestions (see Figure 4, line 3). There exists a rich set of dependency types that can be used with out approach, including method calls, field accesses, method overriding, and textual similarity. We chose to implement our first prototype for only the method call and field access relations for specific reasons. First, this is our first experience with this idea and we wanted to keep the approach as simple as possible to be able to study its properties and to manually reason about the results. We chose method calls and field accesses as the two most relevant

relation types based on a previous empirical study of software investigation using Eclipse [Robillard et al. 2004].

Graph Generation Technology. Suade requires access to a graph of program dependencies to generate suggestions. A variety of approaches can be used to generate this graph. In our initial prototype [Robillard 2005], we used a standalone fact extractor to generate a program graph (this fact extractor has now evolved into the plug-in called JayFX). JayFX requires an initial analysis phase, but access to dependency data is instantaneous once the graph has been created. The only major drawback of this approach is that modifications to the source code require re-generating the program database to avoid inconsistencies. To address this problem, we developed a second prototype that could generate the graph directly from Eclipse's underlying dependency model. This second prototype no longer required an explicit graph generation phase. Unfortunately, this prototype proved problematic for three reasons. First, it was not possible to access Eclipse's underlying dependency model through published interfaces, resulting in deployment problems. Second, the information contained in Eclipse's underlying model did not allow us to perform a class hierarchy analysis that was as precise as JayFX's analysis.[12] Finally, it turned out that for a majority of usage scenarios, it was actually faster to recompute the graph with JayFX than to use Eclipse's underlying model. For these reasons, we reverted to JayFX to generate the graph for Suade release 0.0.1.

Semantics of Calls Relation. Another important consideration when designing the static analysis used to compute the dependency relations was the specification of the semantics of the *calls* (and *called by*) relations with respect to virtual calls. Two main alternatives are possible, namely, to consider a calls relation to be between:

(1) the caller and the static method called as determined through type checking.
(2) the caller and all method implementations potentially invoked through dynamic binding.

In the context of our algorithm, both alternatives have advantages and disadvantages. On one hand, using only static types will result in fewer dependencies and has, thus, a better chance of identifying important relations. However, certain related elements may not be identified if they are only accessed through dynamic calls. On the other hand, traversing class hierarchies to infer methods potentially called will elicit more dependencies but, in the case of large class hierarchies making an important use of overriding, this may result in an artificially low level of specificity. To investigate how these factors played out in practice, our initial prototype [Robillard 2005] implemented the two alternative semantics for the *calls* and *called by* relations: to include only static bindings, and to include all potentially called methods as generated using class hierarchy analysis (CHA). Experience with the initial prototype identified the CHA

[12]When analyzing a method call, JayFX determines the static type of the expression on which the method was called, whereas the Eclipse model stores the first declaration for the method that it finds in the class hierarchy.

semantics for the calling relations as clearly superior. Among other reasons, the static binding alternative often led to awkward situations in the case of methods executed only as the result of dynamic binding (in which case no caller would be found). Hence, based on our experience with the initial prototype, we decided to adopt the CHA semantics for Suade.

Inclusion of Library Elements. In a software project, program elements often refer to a number of elements that are not part of the project proper, but are defined in external libraries. An important question for our approach is whether dependencies to library elements should be part of the algorithm's computation (i.e., appear in the S_{forward} and S_{backward} sets of Figure 4). The overall impact of this decision is easy to predict: to include library elements results in a general decrease of specificity. More specifically, project elements that are used in combination with a great number of library calls will see their degree of potential interest decrease as a result of the lowered specificity. To explore how these factors played out in practice, our initial prototype included an option to configure the function returning the range of a relation (lines 9 and 12 of Figure 4) to omit the elements that were not defined in the source code analyzed. This way, library elements that are typically not investigated by developers are left out of the analysis and, in consequence, of the suggestion sets produced. Experimentation with the tool quickly showed that the inclusion of library elements was distracting at best, because these were almost never investigated. To keep Suade as easy to use and configure as possible, we simply eliminated the option to include library elements for release 0.0.1.

6. CASE STUDIES

We present two case studies intended to build a body of evidence that the analysis of topologies in the structural dependencies of a program can help suggest elements of interest to developers investigating source code. We selected our case studies by identifying, in two different target systems, a high-level concern that could reasonably form the object of software investigation in the context of a change task. For each case, we posit a scenario of the use of our technique, and discuss the results obtained. Because these case studies (and the experiment of Section 7) are intended to provide a baseline record of the approach's performance, we focus on applications of the technique on crisp sets (i.e., sets of interest where all the elements have a degree value of 1.00).

6.1 jEdit Study

Our first case study addresses a feature of the jEdit system described in Section 2. For this study, we consider the case of a developer in charge of performing a modification to the implementation of the AUTOSAVE feature. This feature is responsible for automatically saving open file buffers at a user-defined frequency. The AUTOSAVE feature has been extensively studied as part of previous experimentation [Robillard et al. 2004].

In this study, we will take the role of a developer having no prior knowledge of the implementation of AUTOSAVE. To identify a starting point for the investigation of the feature, we first perform a general text search for the string "autosave."

Table I. Analysis Results for the jEdit AUTOSAVE Seed

#	Element	Reason	Degree	Rel.
1	BufferIORequest.run()	Calls autosave, Accesses AUTOSAVE	0.81	Y
2	Autosave.actionPerformed(ActionEvent)	Calls autosave	0.61	Y
3	BufferIORequest.toString()	Accesses AUTOSAVE	0.60	N
4	Buffer.AUTOSAVE_DIRTY	Accessed by autosave	0.58	Y
5	AutosaveBackupOptionPane._init()	Accesses autosave	0.54	Y
6	AutosaveBackupOptionPane._save()	Accesses autosave	0.54	Y
7	Buffer.IO	Accessed by autosave	0.51	S
8	VFSManager.getFileVFS()	Called by autosave	0.51	N
9	BufferIORequest.BufferIORequest(...)	Called by autosave	0.47	S
10	BufferIORequest.write(...)	Called by autosave	0.46	S
11	FileVFS._createOutputStream(...)	Called by autosave	0.46	N
12	UrlVFS._createOutputStream(...)	Called by autosave	0.46	N
13	VFS._createOutputStream(...)	Called by autosave	0.46	N
14	Buffer.autosaveFile	Accessed by autosave	0.45	Y
15	BufferIORequest.session	Accessed by autosave	0.45	N
16	BufferIORequest.vfs	Accessed by autosave	0.45	N
17	BufferIORequest.view	Accessed by autosave	0.45	N
18	Buffer.dirty	Accessed by autosave	0.44	S
19	BufferIORequest.path	Accessed by autosave	0.43	N
20	Buffer.LOADING	Accessed by autosave	0.43	S
21	BufferIORequest.buffer	Accessed by autosave	0.40	N
22	VFSManager.runInWorkThread(...)	Called by autosave	0.35	N
23	Buffer.setFlag(int, boolean)	Called by autosave	0.33	S
24	Buffer.getFlag(int)	Called by autosave	0.31	S
25	WorkRequest.setStatus(String)	Called by autosave	0.31	S
26	Buffer.isDirty()	Called by autosave	0.30	S
27	CBZip2OutputStream.close()	Called by autosave	0.30	N
28	VFS.getFileName(String)	Called by autosave	0.30	N
29	WorkRequest.setAbortable(boolean)	Called by autosave	0.30	N
30	TarOutputStream.close()	Called by autosave	0.28	N
31	jEdit.getProperty(String, Object[])	Called by autosave	0.22	N

Because this query returns 93 matches, we refine it to return only the class members (fields or methods) that match the string "autosave." This query returns the following four elements:

AutosaveBackupOptionPane.autosave
Buffer.autosave()
BufferIORequest.AUTOSAVE
BufferIORequest.autosave()

Instead of manually and iteratively exploring the 31 dependencies to and from these elements, we use Suade to generate a ranked list. Table I presents the complete results of the analysis.

We illustrate the value of Suade's ranking through an assessment of the relevance of each element returned. We performed this assessment as follows. For each element, we answer the question "is this element relevant to a developer trying to understand how buffers are automatically saved in jEdit?" with the qualifiers "relevant" (Y), "not relevant" (N), or "somewhat relevant" (S). The last column of Table 1 (Rel.) indicates our assessment. For example, element #4 is

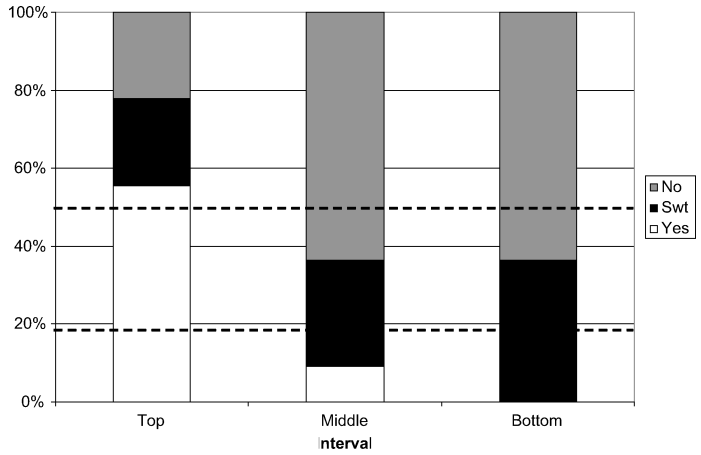

Fig. 11. Ratio of relevant elements for different intervals. The dashed lines represent the overall ratios.

judged relevant (because it is a flag used to indicate that a buffer has not been saved since the last autosave operation). In contrast, element #8 is not judged relevant (because it is a utility function returning a reference to a virtual file system manager).

To synthesize this assessment, we divide the group of 31 suggestions into three groups: the top 9 suggestions (elements 1–9), the middle 11 suggestions (elements 10–20), and the bottom 11 suggestions.[13] We can then calculate the ratio of relevant, somewhat relevant, and irrelevant elements in each group.

Figure 11 shows how the elements (as assessed by the investigator) are distributed in the three intervals (as determined by Suade). In the interval of top suggestions (as generated by Suade), 56% of the suggested elements are relevant (as opposed to 19% if the relevant elements had been uniformly distributed across all three intervals). We can make the opposite observation for the bottom interval. In this case, 64% of elements are not relevant (as opposed to 52% if the irrelevant elements had been uniformly distributed across all three intervals). This exercise shows that, in this case, Suade ranks more relevant elements in the top interval and more irrelevant elements in the bottom interval than would have been expected by chance.

A qualitative inspection of the results explains these overall results. We discuss the detailed topology of only the top and bottom elements as a representative example of both cases. The top element (`BufferIORequest.run()`) is the method in jEdit that calls `autosave()`, and one of only three methods in the entire program that accesses field `AUTOSAVE`. For these reasons, it is very specific to the set of interest, for more than one relation. In addition, one of the accessors of field `AUTOSAVE` (method `Buffer.autosave()` is also in the set of interest, hence an additional degree of reinforcement.

[13]The groups were created to have a cardinality as equal as possible without distributing elements of identical degree across two groups.

In the case of the bottom element (`jEdit.getProperty(String, Object[])`), it is simply a utility method referenced by 55 sites in the program. Because of its very low specificity, it naturally gravitates towards the bottom of the suggestion list.

As this case study illustrates, topological patterns can help estimate the potential relevance of an element to a set of interest. Although, as expected, the technique is not perfectly accurate (e.g., in the case of element #3), our assumption is that overall the topological clues will be strong enough for the technique to be usable. In this case, a developer would have benefited from the technique by being able to search all the dependencies to the set of interest in one operation and to determine immediately the trigger point for the autosave feature from the top suggestion.

6.2 Azureus Study

Our first case study explains how the algorithm can be useful in practice, but is subject to investigator bias. We performed another case study to gather similar evidence that would not be biased in the same way. For this study, we chose to generate a suggestion set intended to help a developer understand the FILE ALLOCATION concern of the Azureus BitTorrent client.[14] In Azureus, disk space for files that are to be downloaded can be allocated using different strategies, and their implementation is scattered across multiple classes. Some of the implementation of the file allocation concern is located in the file `DiskManagerImpl`. This is a large, complex class that has been modified multiple times. As a result, it is a target of choice for our analysis. As our set of interest, we selected all the members of `DiskManagerImpl` that had the word "allocation" or a variant in it. This resulted in a set of one field and three methods. We ran Suade on this initial set of interest. The resulting suggestion set comprised 54 elements.

We then asked two experts to evaluate the results and to qualify each element in the set according to its relevance. The two experts were graduate students who had conducted a detailed analysis of the file allocation concern in Azureus as part of a course project. The experts had performed their analysis using the FEAT concern modeling tool,[15] the SA4J static analysis tool,[16] the JProbe profiler,[17] and manual analysis of the source code.

The experts were asked to look at each element in the suggestion set and answer the question "is this element relevant to a developer trying to understand how files are allocated in Azureus?", using the answers "Yes," "No," and "Somewhat." The experts were unaware of the reason their expertise was required or how the list of elements had been generated. The degree value for each element in the set was not revealed (the list was ordered alphabetically by class name, then member name, of each of the 54 elements in the suggestion set).

Working as a team for over one hour and using the features of Eclipse, the experts produced a qualification of each element in the suggestion set. Out of

[14]http://azureus.sourceforge.net/. Version 2.2
[15]http://www.cs.ubc.ca/labs/spl/projects/feat
[16]http://www.alphaworks.ibm.com/tech/sa4j
[17]http://www.quest.com/jprobe/index.asp

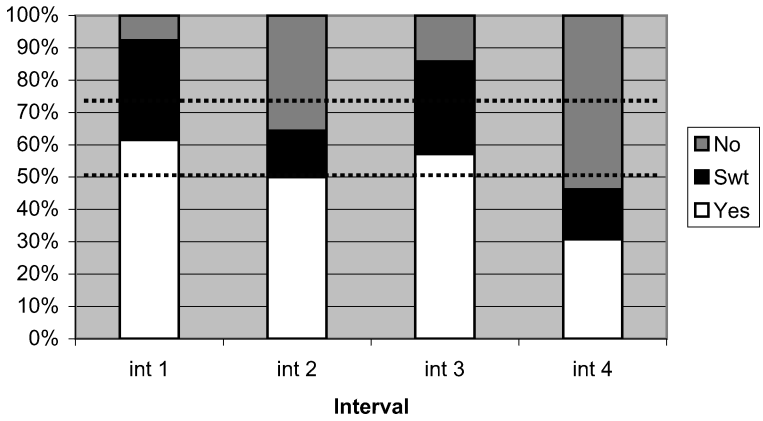

Fig. 12. Ratio of relevant elements for different intervals. The dashed lines represent the overall ratios.

54 elements in the list, 27 (50%) were marked as relevant, 12 (22%) were marked as somewhat relevant, and 15 (28%) were marked as not relevant. Because the experts worked as a team, their classification was consensual and reflected their overall combined knowledge of the system.

We then analyzed whether elements identified as relevant by the experts were associated with high membership degree as generated by our technique, in a way similar to the one described in Section 6.1. To this end, we sorted all elements by descending membership degree and counted the number of relevant, somewhat relevant, and irrelevant elements in different intervals. We separated the sorted list of suggestion into four intervals: the top 13 suggestions, two middle groups of 14 suggestions, and the bottom 13 suggestions. Despite the fact that there were a number of large groups of elements with the same degree value (up to 10), with this classification we were also able to include all the elements with the same degree in the same interval. Figure 12 shows the proportion of relevant, somewhat relevant, and irrelevant elements in each interval. The two horizontal lines mark the overall ratios for the 54 elements (listed above).

From this figure we observe that the approach appropriately ranked the top and bottom elements with a satisfactory degree of accuracy: 62% of the 13 suggestions were relevant (above the overall ratio of 50%), and 54% of the suggestions in the bottom interval are not relevant (again above the overall ratio of 28%). We note that the results for the middle intervals cannot be positively interpreted. However, because the practical use of the approach involves perusing the top suggestions, we are focusing our efforts on validating the top of the suggestion set.

6.3 Experimental Critique

The validity of the evidence gathered as part of our case studies is influenced by a number of factors. We describe the most important of these factors here, along with our efforts to limit them.

First, our results might reflect accidentally unique properties the input sets chosen, as opposed to the more general properties of Java programs. The presentation of two cases from two different systems helps mitigate this risk. In addition, although the code analyzed in both cases revolves around the the high-level concept of a "file," this is a very general theme in software development, and the particulars of the implementation involve mechanisms touching multiple and varied domains, including the graphical user interface, property management, and multithreading (jEdit), and network access and error handling (Azureus).

Second, the interpretation of the jEdit case study is based on a subjective assessment of the relevance of each element generated by our technique. However, the complete list of elements generated is made available, so that independent researchers can interpret our analysis in the light of their own judgment. The evidence provided by the jEdit study is also corroborated by a second case study where independent experts evaluated the relevance of the results of the algorithm. This strategy limits the influence of investigator bias to the selection of the set of interest associated with the file allocation concern. However, this set of interest was obtained by pattern-matching a regular expression and not through an ad-hoc selection. The experts' evaluation is also made public,[18] so that it can be assessed independently. The case studies thus provide reliable evidence that the technique can be useful in realistic conditions.

7. QUANTITATIVE VALIDATION

The two case studies described in the previous section provided initial evidence of the usefulness potential for topology analysis of dependencies. To assess whether these early results could generalize to different software investigation scenarios, we conducted an experiment to evaluate the usefulness of the algorithm's results. We conducted this experiment prior to the release of Suade 0.0.1, and for this reason the results were obtained using a different prototype that relied on Eclipse's Java model for computing structural dependencies. Although the different implementations do not always produce identical suggestions sets (see Section 5.3), we do not expect the overall conclusions of this experiment to be sensitive to this implementation detail given that it involves the application of our algorithm on 300 distinct input sets.

7.1 Study Design

In this experiment, we investigated how our technique could help discriminate between relevant and irrelevant elements in different *benchmark concerns*. In the context of this experiment a benchmark concern (or benchmark, for short) is a number of fields and methods in a system that are associated with the implementation of a concern of interest to a developer. Given a benchmark, we can evaluate our recommendation technique by studying how successful it is at suggesting benchmark elements given a subset of the benchmark as a set of interest.

[18]http://www.cs.mcgill.ca/~martin/esecfse2005/

Using the terminology of Section 4.2, we formalize our experiment as follows.

(1) **Target program.** Choose a target program $P = (E, R)$.
(2) **Benchmark.** Given a program $P = (E, R)$, determine a benchmark $B \subseteq E$. The benchmark should include elements that are related to the implementation of a clearly-defined high-level concern that could realistically be associated with a software modification task.
(3) **Samples.** Given a program $P = (E, R)$ and a benchmark B, randomly select n samples I_i of p elements such that $\forall\, I_i, I_i \cap B \neq \emptyset$. The samples I_i will be used as sets of interest. Although, in most cases, it is reasonable to have samples formed only of benchmark elements ($I_i \subseteq B$), a more rigorous evaluation of the technique should also consider sets of interest containing noise. We characterize the *relevance* of a sample as $r = |I_i \cap B|$.
(4) **Application.** For each sample I_i, generate a suggestion set $\tilde{S}_i$.
(5) **Selection window.** Determine a selection window w of top suggested elements to consider (e.g., only the element with highest degree, top three elements, etc.).
(6) **Data.** Determine the probability p_R of selecting a benchmark element by randomly choosing an element in the suggestion set, and the probability p_F of choosing a benchmark element by randomly choosing an elements in the filtered suggestion set F_w containing the top w suggested elements. We have $p_R = \frac{|B \cap \tilde{S}_i|}{|\tilde{S}_i|}$ and $p_{F_w} = \frac{|B \cap F_w|}{w}$.
(7) **Analysis.** Using a statistical technique, determine whether, over all n samples, p_{F_w} is significantly greater than p_R.

As can be seen from this methodology our experiment can be parameterized in terms of program (P), benchmark (B), number of samples (n), sample cardinality (p), sample relevance (r), and selection window (w). We call a fixed combination of these parameters a *configuration*.

Our general research hypothesis is that our technique can help a developer find relevant elements to investigate. In the context of our experiment, our research hypothesis is that $p_F > p_R$ which we can interpret as "in the absence of a better clue, a developer has a better chance of finding a benchmark element by looking at top suggestions than randomly inspecting the results of cross-reference searches on elements of interest." The null hypothesis is that looking at highly ranked elements does not significantly improve one's chance of finding a benchmark element.

7.2 Study Configurations

To ensure a reasonable level of external validity for our study, we instantiated the experiment for 45 different configurations. In the following, we discuss our choice of parameters.

Programs and Benchmarks. We ran our experiment on four open-source Java programs: Violet,[19] a small application to draw diagrams in the Unified

[19] www.horstmann.com/violet/

Table II. Characteristics of Target Programs and Benchmarks

System	LOC	Benchmark	B. Types	B. Elements
Violet 0.15	6,744	CONNECTIONS	5	9
LOCC 3.3	27,430	OUTPUT	29	176
JHotDraw 5.3	13,581	ARROW-1	5	11
JHotDraw 5.3	13,581	ARROW-2	7	30
jEdit 4.2-final	88,294	MARKER	8	45

Modeling Language (UML); LOCC,[20] a command-line application to extract metrics from software systems; JHotDraw, the default drawing application built on the JHotDraw application framework (see Section 4.6); jEdit, a full-featured text editor (see Section 2). We selected these target systems because they offered a reasonable size and complexity spectrum, because the set offered a comparison between both similar systems (e.g., Violet and JHotDraw) and different systems (e.g., LOCC and jEdit), and because we were able to obtain satisfactory benchmarks from these systems.

Table II shows the basic characteristics of each program and a brief description of the benchmark concerns defined on them. The CONNECTION benchmark defined on Violet was produced independently by a separate researcher designing an experiment for a different project. In the view of the researcher, this benchmark corresponds to the code that one must understand to fix a small issue that allows users of Violet to create invalid connections in a UML diagram. The OUTPUT benchmark on LOCC was created independently by a second researcher interested in studying code clones in LOCC. This large concern corresponds to the implementation of the output functionality of LOCC, which supports different output formats (e.g., text, CSV). The two ARROW concerns of JHotDraw were produced by the author. They correspond to the code that supports the adornment of lines with arrow ends. The first version of this benchmark omits a class (`ArrowTip`) which very neatly encapsulates the code implementing the graphical line decorations. The second version includes the entire class and an additional method that calls its constructor. This benchmark exists in two versions because, given the design of our experiment, the addition of an entire class in a small benchmark can have an important impact on the results. Having both versions of the benchmark allows us to study this impact. We discuss this phenomenon in more detail in Section 7.3. Finally, the MARKER concern was produced by the author as part of a different, unrelated empirical study of concern evolution (see Section 2). The concern description was reused as is from the other study. Taken together, this collection of five benchmarks covers a realistic spectrum of benchmark sizes and styles. Specifically, it includes benchmarks ranging from small and focused (CONNECTIONS, ARROW-1) to large and general (OUTPUT).

Number of Samples. The maximum number of samples for an experiment is bounded by the number of potential combinations of elements in a benchmark, given a sample cardinality. For example, with a sample cardinality of 2

[20] csdl.ics.hawaii.edu/Plone/research/locc.

Table III. Experimental Configurations

Name	Symbol	Values
Benchmark	B	{Connections, Output, Arrow-1, Arrow-2, Marker}
Number of samples	n	{20}
Sample cardinality and relevance	(p, r)	{(2, 2),(3, 3),(4, 3)}
Selection window	w	{1, 3, 5}

the VIOLET concern offers a maximum of $\binom{9}{2} = 36$ different samples. As for a lower bound, we must choose enough samples to enable the use of statistical techniques on the data collected. For simplicity of comparison, we chose $n = 20$ for all our configurations.

Sample Cardinality and Relevance. In terms of sample cardinality and relevance, we chose three sets of (p, r) parameters that realistically represent a small but nonunit starting set for a program investigation session: (2, 2), (3, 3), and (4, 3). The first two sets represent samples of zero nonbenchmark elements and two and three benchmark elements, respectively. The last set of parameters characterizes samples consisting of three benchmark elements and one non-benchmark element. One must note that including non-benchmark elements in the sample raises the question of how to select such elements. To solve this question we created our (4, 3) samples by generating a (3, 3) sample, applying the suggestion algorithm, and selecting the fourth element from the suggestions that were not in the benchmark. This strategy is intended to represent a user including an element that is "reasonable, but not quite right" in the set of interest.

Selection Window. Hypothesizing that developers do not like to scroll down long lists of potential results, we aggressively restricted the selection window for our experiment, choosing the values of 1, 3, and 5. A small window size also helps us achieve a more robust interpretation of the results because the more w increases, the more similar F_w and $\bar{S}_i$ become.

7.3 Results

Table III summarizes the parameter values we chose for our experimental configurations. We applied our suggestion algorithm to 20 samples for each configuration and recorded, for each:

—The size of the suggestion set ($|\bar{S}_i|$);
—The number of suggestions in the benchmark ($|\bar{S}_i \cap B|$);
—Whether the top suggestion is a benchmark element (for $w = 1$);
—The number of benchmark elements in the top 3 suggestions (for $w = 3$);
—The number of benchmark elements in the top 5 suggestions (for $w = 5$);

For the last three items, whenever there existed a tie for a top position that could have impacted the result, we broke the tie by randomly (and automatically) choosing the top elements among the equal-valued suggestions. The completion of these experiments produced 45×20 (p_R, p_F) pairs.

Table IV. Analysis Results

		$w=1$			$w=3$			$w=5$		
Benchmark	(p,r)	OR	95%L	95%U	OR	95%L	95%U	OR	95%L	95%U
Arrow-1	(2,2)	4.3	2.0	9.2	5.0	2.6	9.7	3.5	2.1	5.6
Arrow-1	(3,3)	13	4.8	36	9.4	4.3	21	6.1	3.9	9.6
Arrow-1	(4,3)	15	6.0	40	11	5.4	21	6.9	4.5	11
Arrow-2	(2,2)	3.4	1.4	8.3	3.0	1.6	5.5	*	*	*
Arrow-2	(3,3)	12	4.4	33	5.7	2.7	12	2.8	1.7	4.7
Arrow-2	(4,3)	4.0	1.6	10	2.6	1.6	4.2	2.2	1.5	3.3
Connections	(2,2)	**0.96**	**0.32**	**2.9**	2.5	1.6	3.9	2.5	1.8	3.4
Connections	(3,3)	**1.7**	**0.61**	**4.6**	2.9	2.0	4.3	3.4	2.5	4.4
Connections	(4,3)	**0.74**	**0.17**	**3.3**	2.4	1.4	4.1	2.8	1.8	4.2
Marker	(2,2)	*	*	*	3.8	2.7	5.3	*	*	*
Marker	(3,3)	9.1	3.9	22	4.9	3.5	6.9	3.2	2.4	4.4
Marker	(4,3)	27	7.7	96	8.3	4.6	15	6.7	4.1	11
Output	(2,2)	**2.0**	**0.80**	**5.0**	2.3	1.6	3.2	1.6	1.3	2.0
Output	(3,3)	2.4	1.1	5.0	3.4	2.1	5.4	2.8	1.9	4.1
Output	(4,3)	**0.48**	**0.20**	**1.1**	**1.0**	**0.63**	**1.6**	**1.2**	**0.75**	**2.0**

7.4 Analysis

For each configuration, we modeled the odds of getting a benchmark element using logistic regression. To account for a potential correlation between observations from the same sample, we used the method of Generalized Estimating Equations (GEE) as implemented in PROC GENMOD [Allison 1999].

Table IV shows the results of the logistic regression. For each configuration, we report the estimated odds ratio (OR), along with the lower (L) and upper (U) bounds of the 95% confidence interval (with two significant digits). This estimate is based on the analysis of the 20 samples generated for a configuration. An OR value is the ratio of the odds of selecting a benchmark element randomly from the top w suggested elements over the odds of selecting a benchmark element randomly from a suggestion set.[21] For example, for ARROW-1, $(2, 2)$, $w = 1$, the odds of choosing a benchmark element by selecting the top suggestion are estimated to be 4.3 times higher than by randomly choosing a suggestion, with a confidence interval of [2.0, 9.2]. The estimated odds ratio is statistically significant if the 95% confidence interval does not include the unit ratio. In three cases, the statistical procedure could not complete the estimation. These cases are reported with a set of asterisks in the table.

As a first synthesis, we can simply count the number of experimental configurations where our suggestion technique was useful. We consider that the technique was useful for configurations where the odds ratio is greater than 1.0 with 95% confidence. For clarity, we have highlighted the results in Table IV where this is **not** the case. Counting cases where we have no data as nonuseful configurations, we conclude that the technique significantly improved the odds of selecting a benchmark element in 35 of the 45 configurations. The most successful configuration is $(p, r, w) = (4, 3, 1)$ for the MARKER concern (OR = 27) and the least successful significant configurations are $(p, r, w) = (4, 3, 3)$ for the

[21]For a specific sample we have $\text{OR} = p_{F_w}(1 - p_R)/p_R(1 - p_{F_w})$.

CONNECTIONS concerns and $(p, r, w) = (3, 3, 1)$ for the OUTPUT concern, both with OR = 2.4.

A closer look at the data allows us to make a number of interesting observations that translate in knowledge directly applicable by users of the technique.

—Using a set of interest of three relevant elements yielded better results than using a set of interest of two relevant elements in all cases. We hypothesize that this phenomenon is caused by the higher number of suggestions generated with three elements (which increase the denominator of p_R), and by the higher potential for reinforcement offered by a three-element set.

—Increasing w tends to decrease OR and reduce the range of the confidence interval. In other words, the variability decreases as we consider more of the top elements. Based on these results, we recommend to consider the 3–5 top suggestions.

—It is difficult to predict the impact of introducing noise in the set of interest. This situation is modeled with the $(p, r) = (4, 3)$ parameters in our experiment. If we compare the results using (3, 3) versus (4, 3) across benchmarks, we see that for ARROW-1 and MARKER the introduction of a noisy element actually improved the OR, whereas for the other benchmarks it had the opposite effect. In the case of OUTPUT, the introduction of a noisy element even rendered the technique useless.

—The ARROW-1 benchmark produced higher OR than the ARROW-2 benchmark for all nine other possible combinations of parameters. This phenomenon can be partially explained as an artifact of our experiment. In the ARROW-2 benchmark, one class (`ArrowTip`) is entirely added to the benchmark as it is entirely related to the concern. Because of our sampling procedure, only a fraction of the elements in `ArrowTip` can be part of the sample set of interest. In this case, our sample does not represent a realistic set of interest because a developer having identified `ArrowTip` as entirely relevant to a concern would normally include the entire class in the set of interest. The impact of selecting a subset of `ArrowTip` is that the other members of the `ArrowTip` class will easily be regenerated as suggestions because there are typically many structural relations among the elements of a class. Because all the elements of `ArrowTip` are part of the benchmark, we can expect the probability p_R to be artificially high, and thus to have a lower OR. This analytic interpretation is confirmed by our comparison of the estimated ORs for the two ARROW benchmarks. The lesson we take away from this observation is that if all the elements of a class implement a concern, they should all be added to the set of interest because they will otherwise form obvious clutter in the suggestion set.

7.5 Experimental Critique

As is the case for most controlled experiments, our quantitative assessment of our recommendation technique was performed in a restricted and synthetic context, and as such important factors must be considered when interpreting the results. We orient our discussion in terms of four common characteristics used

to assess experimental designs: construct validity, internal validity, external validity, and reliability [Yin 1989, p. 33].

The *construct validity* of our study (or correctness of our operational measures) is affected by the fact that not all samples necessarily represent sets of interest that developers would use. To derive an objective and quantitative measurement of the success of the technique, we needed to avoid any subjective discrimination between samples. In practice, users of the technique may have different personal styles for constructing sets of interest, which may affect the results. Our measurement of the success of the technique also involves an imperfect approximation of the use of the technique. Specifically, although we assess the technique in terms of how it improves the odds of randomly selecting a benchmark element from a set of structurally related candidates, this is only a model because, in practice, we do not expect users to choose randomly. Instead, we can assume that users will choose suggestions based on keywords, previous knowledge of the task, intuition, etc. However, we feel that our random model is appropriate as it estimates the uncertainty associated with choosing elements from a list while making as few assumptions as possible about the selection strategy used.

The main strength of this experimental design is its high *internal validity* (or soundness of the relationship between independent and dependent variables). Because all the factors potentially affecting the difference between p_F and p_R are under our direct control, any significant difference must be caused by the ranking produced by our technique.

External validity establishes to what extent the results of the study can be generalized. As can be seen from Table IV, the success level of the technique is affected by the benchmark and by the nature of the set of interest. Given the unbounded variety of software and programming tasks, we can expect that there will be situations where the technique will not be useful. However, given that we were able to confirm the success of the technique for 35 configurations involving five different benchmarks defined on four different systems by three different programmers, we expect that additional experimentation with the current form of the technique should lead to similar success levels.

Finally, *reliability* qualifies to which degree the study can be repeated with the same results. In our case, we conducted all the experiments on open-source systems using procedures and algorithms completely described in this article and the benchmarks are available on request from the author. In addition, the direct interpretation of the statistical data does not involve a subjective, a posteriori assessment of relevance. For these reasons, it should be possible to independently replicate our results.

8. CONCLUSION

We presented a technique for automatically suggesing elements of potential interest to a developer involved in a program investigation task. Our technique is based on an analysis of the topology of a graph of structural dependencies for a software system. The technique takes as input a fuzzy set representing

elements of interest to a developer and produces a fuzzy set of related elements, whose degree of membership is calculated by analyzing how specific an element is to the set of interest, and how its relation to the set of interest is reinforced by existing relations with other elements in the set of interest. The intuition behind our technique is that analyzing the topological properties of the structural dependencies of a software system can help determine the potential for an element to be worthy of detailed investigation by a developer. A qualitative study of the results produced for two sets of interest describing useful concepts in medium-size systems shows how our algorithm can help developers quickly select program elements worthy of investigation while avoiding less interesting ones. A quantitative experiment of the success of the technique in 45 situations involving five different program investigation scenarios defined on four different systems showed that the technique can significantly increase the odds of identifying a relevant element over unguided investigation of the code in at least 35 of our 45 experimental configurations. We conclude that topology analysis appears to be a promising and cost-effective way to help developers navigate source code.

ACKNOWLEDGMENT

The author is grateful to José Correa of the McGill University Statistical Consulting Service for his professional advice and dilligent help with the analysis of the data for the quantitative experiment. Many thanks also go to Frédéric Weigand Warr for his work on Suade, and to Félix Martineau, Philippe Nguyen, Imran Majid and Ekwa Duala-Ekoko for contributing to the empirical evaluation of the work. This article has greatly benefited from the valuable comments of Davor Čubranić, Nomair Naeem, Rob Walker, the members of the Software Practices Lab at UBC, and the anonymous ESEC/FSE 2005 and TOSEM reviewers.

REFERENCES

AGRAWAL, H. AND HORGAN, J. R. 1990. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 246–256.

AHO, A. V. 1980. Pattern matching in strings. In *Formal Language Theory: Perspectives and Open Problems*, R. V. Book, Ed. Academic Press, 325–347.

ALLISON, P. D. 1999. *Logistic Regression Using the SAS system—Theory and Application*. SAS Institute Inc.

ANTONIOL, G., CANFORA, G., DE LUCIA, A., AND MERLO, E. 1999. Recovering code to documentation links in OO systems. In *Proceedings of the 6th IEEE Working Conference on Reverse Engineering*. 136–144.

ANTONIOL, G. AND GUÉHÉNEUC, Y.-G. 2005. Feature identification: A novel approach and a case study. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*. 357–366.

BOEHM, B. W. 1976. Software engineering. *IEEE Trans. Comput. 12*, 25, 1226–1242.

CHEN, Y.-F., NISHIMOTO, M. Y., AND RAMAMOORTHY, C. 1990. The C information abstraction system. *IEEE Trans. Softw. Engin. 16*, 3, 325–334.

EISENBARTH, T., KOSCHKE, R., AND SIMON, D. 2003. Locating features in source code. *IEEE Trans. Softw. Engin. 29*, 3, 210–224.

EISENBERG, A. D. AND DE VOLDER, K. 2005. Dynamic feature traces: Finding features in unfamiliar code. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*. 337–346.

GALLAGHER, K. B. 1996. Visual impact analysis. In *Proceedings of the IEEE International Conference on Software Maintenance*. 52–58.

GALLAGHER, K. B. AND LYLE, J. R. 1991. Using program slicing in software maintenance. *IEEE Trans. Softw. Engin. 17*, 8, 751–761.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1995. *Design Patterns—Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley Longman, Inc.

GOLDBERG, A. 1984. *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley.

GYIMÓTHY, T., ÁRPÁD BESZÉDES, AND FORGÁCS, I. 1999. An efficient relevant slicing method for debugging. In *Proceedings of the 7th European Software Engineering Conference and 7th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*. Lecture Notes in Computer Science, vol. 1687. Springer-Verlag, Berlin, Germany, 303–321.

JACKSON, D. AND ROLLINS, E. J. 1994. A new model of program dependence for reverse engineering. In *Proceedings of the 2rd ACM SIGSOFT Symposium on the Foundations of Software Engineering*. 2–10.

JANZEN, D. AND DE VOLDER, K. 2003. Navigating and querying code without getting lost. In *Proceedings of the 2nd ACM International Conference on Aspect-Oriented Software Development*. 178–187.

KERSTEN, M. AND MURPHY, G. C. 2005. Mylar: a degree-of-interest model for IDEs. In *Proceedings of the 4th ACM International Conference on Aspect-Oriented Software Development*. 159–168.

KRUCHTEN, P. 2000. *The Rational Unified Process: An Introduction* 2nd Ed. Addison-Wesley.

LEJTER, M., MEYERS, S., AND REISS, S. P. 1992. Support for maintaining object-oriented programs. *IEEE Trans. Softw. Engin. 18*, 12, 1045–1052.

MARCUS, A., SERGEYEV, A., RAJLICH, V., AND MALETIC, J. I. 2004. An information retrieval approach to concept location in source code. In *Proceedings of the 11th IEEE Working Conference on Reverse Engineering*. 214–223.

OBJECT TECHNOLOGY INTERNATIONAL, INC. 2001. Eclipse platform technical overview. White Paper.

ORSO, A., APIWATTANAPONG, T., LAW, J., ROTHERMEL, G., AND HARROLD, M. J. 2004. An empirical comparison of dynamic impact analysis algorithms. In *Proceedings of the 26th ACM/IEEE International Conference on Software Engineering*. 491–500.

ORSO, A., SINHA, S., AND HARROLD, M. J. 2001. Incremental slicing based on data-dependences types. In *Proceedings of the IEEE International Conference on Software Maintenance*. 158–167.

ROBILLARD, M. P. AND WEIGAND WARR, F. 2005. ConcernMapper: Simple view-based separation of scattered concerns. In *Proceedings of the Eclipse Technology Exchange Workshop at OOPSLA*. 65–69.

ROBILLARD, M. P. 2005. Automatic generation of suggestions for program investigation. In *Proceedings of the Joint 10th European Software Engineering Conference and 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering*. 11–20.

ROBILLARD, M. P. 2006. Tracking concerns in evolving source code: An empirical study. In *Proceedings of the 22nd IEEE International Conference on Software Maintenance*. 479–482.

ROBILLARD, M. P., COELHO, W., AND MURPHY, G. C. 2004. How effective developers investigate source code: An exploratory study. *IEEE Trans. Softw. Engin. 30*, 12, 889–903.

SALAH, M. AND MANCORIDIS, S. 2004. A hierarchy of dynamic software views: from object-interactions to feature-interactions. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*. 72–81.

SANELLA, M. 1983. *The Interlisp-D Reference Manual*. Xerox Corporation.

SILLITO, J., MURPHY, G. C., AND DE VOLDER, K. 2006. Questions programmers ask during software evolution tasks. In *Proceedings of the 14th ACM SIGSOFT Symposium on the Foundations of Software Engineering*. 23–33.

SNELTING, G. 1998. Concept analysis—a new framework for program understanding. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. 1–10.

TIP, F. 1995. A survey of program slicing techniques. *J. Program. Lang. 3*, 3, 121–189.

TONELLA, P., ANTONIOL, G., FIUTEM, R., AND MERLO, E. 1997. Variable precision reaching definitions analysis for software maintenance. In *Proceedings of the 1st IEEE Euromicro Conference on Software Maintenance and Reengineering*. 60–67.

WEIGAND WARR, F. AND ROBILLARD, M. P. 2007. Suade: Topology-based searches for software investigation. In *Proceedings of the 29th ACM/IEEE International Conference on Software Engineering*. 780–783.

WEISER, M. 1984. Program slicing. *IEEE Trans. Softw. Engin. 10*, 4, 352–357.

WILDE, N. AND SCULLY, M. C. 1995. Software reconnaissance: Mapping program features to code. *Softw. Mainten. Resear. Prac. 7*, 49–62.

WONG, W. E., GOKHALE, S. S., HORGAN, J. R., AND TRIVEDI, K. S. 1999. Locating program features using execution slices. In *Proceedings of the IEEE Symposium on Application-Specific Systems and Software Engineering and Technology*. 194–203.

YIN, R. K. 1989. *Case Study Research: Design and Method* 2nd Ed. Applied Social Research Methods Series, vol. 5. Sage Publications Ltd., UK.

YING, A. T., MURPHY, G. C., NG, R., AND CHU-CARROLL, M. C. 2004. Predicting source code changes by mining change history. *IEEE Trans. Softw. Engin. 30*, 9, 574–586.

ZHAO, W., ZHANG, L., LIU, Y., SUN, J., AND YANG, F. 2004. SNIAFL: Towards a static noninteractive approach to feature location. In *Proceedings of the 26th ACM/IEEE International Conference on Software Engineering*. 293–303.

ZIMMERMANN, H.-J. 1996. *Fuzzy Set Theory and Its Applications,* 3rd ed. Kluwer Academic Publishers, The Netherlands.

ZIMMERMANN, T., WEIßGERBER, P., DIEHL, S., AND ZELLER, A. 2004. Mining version histories to guide software changes. In *Proceedings of the 26th ACM/IEEE International Conference on Software Engineering*. 563–572.

Received March 2006; revised February 2007, April 2007; accepted August 2007