Automatic Generation of Suggestions for Program Investigation

Martin P. Robillard

School of Computer Science
McGill University
Montreal, QC, Canada
martin@cs.mcgill.ca

ABSTRACT

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 an inherently 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 typically hundreds of dependency paths found in a non-trivial program. In this paper, we propose 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—*Restructuring, reverse engineering, and reengineering*

General Terms

Algorithms, Experimentation, Human Factors

Keywords

Static analysis, feature location, structural program dependencies

1. INTRODUCTION

Software projects typically go through multiple iterations during their lifetime [13], with many iterations involving a number of modifications to the source code. As part of most software modification tasks, a developer must investigate the source code associated with the task prior to modifying it [4].

ESEC-FSE'05, September 5–9, 2005, Lisbon, Portugal.

In essence, the goal of investigating source code in the context of a software modification task is to solve an instance of the *concept assignment problem* [3], namely, to identify how high-level concepts associated with the task are implemented in source code. For example, a developer asked to solve an I/O performance problem may need to discover and understand the code implementing a buffering algorithm.

The concept assignment problem is a hard problem in software engineering because it is an inherently human activity, whose success depends on a large number of unpredictable factors, such as intuition and luck. In a recent empirical study of program investigation [15], we observed that a distinctive characteristic of effective developers was their tendency to investigate source code by following structural dependencies. While this practice can help make program investigation more systematic, it does not solve the concept assignment problem. Indeed, in any non-trivial 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 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 [15].

This problem can be mitigated through approaches that automatically provide developers with an estimate of the code relevant to a concept, concern, or feature (see Section 2). This paper provides a contribution to this corpus by investigating the hypothesis that *it is possible to guide developers towards interesting sections of code by analyzing the topology of a program's structural dependencies*. 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 would be inexpensive enough to use in a highly iterative fashion and on incomplete or incorrect programs.

As part of our investigation we developed an algorithm for a static analysis that calculates program elements of likely 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 element suggested 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 a prototype of our algorithm for Java systems and basic tool support allowing developers to use it for software

evolution tasks. Using this prototype, we conducted an experiment to study the quantitative nature of the results produced and their stability in the face of different configurations and inputs. We also performed two case studies on medium-size systems where we qualitatively evaluated the suggestions produced. Our results show that the algorithm is stable and produces suggestions that can help a developer rapidly build a core set of program elements associated with a task while avoiding code that is not relevant. We conclude that analyzing the topology of a program's structural dependencies is a promising technique for helping developers navigate source code efficiently.

In the rest of this paper, we first present an overview of techniques previously proposed to help developers investigate source code (Section 2). We then present our algorithm (Section 3), and describe its current implementation for Java (Section 4). We report on the empirical evaluation of our algorithm in Section 5 (quantitative experiment) and in Section 6 (case studies). Finally, we discuss the results of the evaluation and additional applications for the technique in Section 7 and present our conclusions in Section 8.

2. RELATED WORK

A variety of approaches have been proposed to help developers identify the source code that may be related to a change. Such approaches usually come under the banner of *concept*, *concern*, or *feature location* approaches, and use a wide range of analysis techniques. Since a survey of all the work in this area is neither possible nor desirable in the context of this paper, we focus on a general categorization of the main approaches, which we illustrate with references to recent work.

2.1 Program Slicing

Program slicing denotes a type of analysis intended to identify the parts of a program that may affect the values computed at some point of interest [19]. Slicing was originally defined as a static analysis technique [21], but dynamic variants have since been developed. For software evolution activities, slicing can be used to help determine the impact of changes [9]. Visual techniques have also been developed to help in this process [8].

Although they are conceptually appealing techniques, static slicing and its variants suffer from practical limitations. First, computing slices can be expensive [21], and pragmatic considerations may require lower-precision data-flow analyses [20]. Second, because a statement is often transitively dependent on many other statements, slices are often very large [11, 21], which limits their usefulness to developers wishing to focus on code of immediate interest.

2.2 Dynamic Analysis

The Software Reconnaissance technique developed by Wilde et al. identifies features in source code based on a analysis of the execution of a program [22]. Software Reconnaissance determines the code implementing a feature 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. [7]. Eisenbarth et al. produce the mapping between components and test cases using mathematical concept analysis (a partial ordering and clustering technique [18]). 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 program analyzed. This step helps achieve a more precise and complete description of the code implementing a feature, at the cost of additional effort for developers using the technique.

Dynamic slicing [1, 10] 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.

In contrast to static approaches, dynamic feature location 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 addition, dynamic approaches 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.

2.3 Information Retrieval

Another approach taken to identify the code associated with a feature is to use information retrieval techniques. Antoniol et al. 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 [2]. 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.

The SNIAFL technique of Zhao et al. [24] 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.

2.4 Repository Mining

A number of approaches can help developers identify elements of interest in the context of a software modification tasks through analysis of a repository of software artifacts. Both Zimmermann et al. [26] and Ying et al. [23] 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 tradeoff of these approaches is the necessity to have 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.

2.5 Static Dependency Analysis

Most techniques proposed to address the concept assignment problem include some form of static dependency analysis, from tool-assisted traversal of dependency paths [5] to automatic searches for dependencies to elements currently active in an integrated development environment [12]. Many of the techniques described in this section partially involve an analysis of structural dependencies. In the space of purely static analysis-based 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. ALGORITHM

Our algorithm for suggesting elements to examine during a program investigation task 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.[1] 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, a value indicating its potential interest to the developer.

The general hypothesis underlying our algorithm is 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. Specifically, two simple but interacting intuitions guide the design of our algorithm:

- **Specificity.** An element y is specific to a set of interest $\bar{I}$ if any element in $\bar{I}$ related to y is related to few elements besides y, and if y itself is related to few elements.
- **Reinforcement.** An element y is reinforced by a set of interest $\bar{I}$ if most elements related to y are in $\bar{I}$.

For example, if a method $m_1 \in \bar{I}$ is only called by a single other method m_2, and m_2 itself does not call any method besides m_1, then m_2 will be highly specific (and potentially interesting to a developer). Our definition of specificity is motivated by the hypothesis that elements that are very specific to a set of interest probably contribute to the implementation of the concept or task associated with the set of interest.

Reinforcement is orthogonal to specificity and can potentially compensate for it. For example, if m_1 is called by 25 methods, 24 of which are also in $\bar{I}$, then the remaining method will be highly reinforced, and thus potentially interesting. Our definition of reinforcement is motivated by the hypothesis that if most elements sharing some structural property are related to a set of interest, it may be desirable (and the developer may be intending) to investigate all of the elements with this property.

In reasoning about specificity and reinforcement it is useful to distinguish between *direct* and *transpose* (inverse) relations. We illustrate this concept for specificity in Figure 1. In the figure, elements A and B are in a set of interest. The specificity for element 1 takes into account that element A relates to two elements (elements 1 and 2, direct relation) and that element 1 is related to no element other than A (transpose relation). The specificity for element 2 also takes into account element A's relation to elements 1 and 2 (direct) but also the fact that element 2 is related to element 4 (transpose). For this reason element 2 is less specific than element 1. In the case of element 3, its specificity is determined by the fact that it is the only element related to B (direct), but itself is related to three other elements (transpose). We use direct and transpose relations in the calculation of both specificity and reinforcement (see Section 3.3 for details).

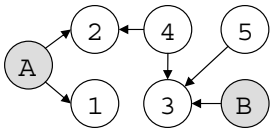

Figure 1: Sample program graph to illustrate specificity

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

3.1 Basic Concepts

Since program investigation is an imperfect process, and since 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 tool supporting our algorithm. In our application of fuzzy set theory, we use notation and definitions consistent with the presentation of Zimmermann [25]. In particular, set variables with an overbar distinguish fuzzy sets from normal (crisp) sets.

Our algorithm relies on the concepts described 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 between these elements.

DEFINITION 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 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 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, i.e., $r \in R \rightarrow r^\top \in R$.*

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

DEFINITION 4 (SET OF INTEREST). *Given a program $P = (E, R)$, a set of interest $\bar{I} = \{(e, \mu_{\bar{I}}(e)) \mid e \in E\}$ is defined as a fuzzy set with membership function $\mu_{\bar{I}}$.*

DEFINITION 5 (SUGGESTION SET). *Given a program $P = (E, R)$, a suggestion set $\bar{S} = \{(e, \mu_{\bar{S}}(e)) \mid e \in E\}$ 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 [25]. For example: $\mu_{\bar{I}} = \{(e_1, 0.5), (e_2, 0.7)\}$.

3.2 Main Algorithm

Figure 2 presents the abstracted analysis algorithm. For each relation type considered, the algorithm calculates a suggestion set based on the relation type.[2] For example, using $l =$ *calls* will generate a suggestion set based on the analysis of the methods called by methods in $\bar{I}$.

Instead of merging the fuzzy sets obtained for each relation using the standard union operator for fuzzy sets,[3] we define a new operator $\uplus$ that works slightly differently: if an element x is in the intersection of both fuzzy sets, the resulting membership degree is

[2]The set of relation types is a crisp set (no overbar).

[3]For two fuzzy sets $\bar{S}_1$ and $\bar{S}_2$ with membership functions $\mu_{\bar{S}_1}(x)$ and $\mu_{\bar{S}_2}(x)$ we usually have $\mu_{\bar{S}_1 \cup \bar{S}_2}(x) = \max(\mu_{\bar{S}_1}(x), \mu_{\bar{S}_2}(x))$.

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

Figure 2: Main algorithm

higher than both maximums, and calculated using the following function:

$$\mu_{\bar{S}_1 \uplus \bar{S}_2}(x) = \max(\mu_{\bar{S}_1}(x), \mu_{\bar{S}_2}(x))^{\frac{1}{1+\min(\mu_{\bar{S}_1}(x), \mu_{\bar{S}_2}(x))}} \quad (1)$$

We designed our merge function (equation 1) to be symmetrical, to have a range between 0 and 1 (inclusive), and to always be greater than 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 an element $(x, 0.50) \in \bar{S}_2$ will result in an element with membership 0.83 in the merged set. The properties of our merge function are obtained by taking a normalized degree value to a power between 0 and 1 (inclusive). Symmetry is achieved through the use of the *min* and *max* functions.

3.3 Analyzing Relations

The function *analyzeRelation* is specified in Figure 3. For each element x in a set of interest (line 8), this function obtains the range of relation l corresponding to domain x (line 9). For example, for $l =$ *calls*, the set S_b contains all the methods called by x. Then, each range element $s \in S_b$ that is not already in the set of interest (lines 10–11) is added to the suggestion set (line 13) with a membership degree that is calculated (line 13) by taking into account the specificity and reinforcement of the element s for both its relation l and its transpose (line 12), by multiplying this value by the degree of the element x being analyzed for dependencies, and by taking the resulting value to the power of α.[4] The parameter α is used to adjust the sensitivity of the algorithm by taking a normalized degree value to a power between 0 and 1. The effect of this operation is to increase the overall degree value and to decrease differences between the degree of elements in a suggestion set. Use of an exponent ensures that the result remains a normalized degree value (i.e., between 0 and 1). The impact of α on the results is reported in detail in Section 5.

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. For example, let us assume that an element x is called by 200 methods. If one of these 200 methods only calls x and no other methods, then it is probably more interesting than the 199 other callers because it is very specific to x. Performing our analysis on the transpose of each relation allows us to factor in this intuition.

The *degree* (*deg*) of the range of a relation is defined in equation 2. This equation is designed to account for the basic tradeoff between specificity and reinforcement. The greater the number of elements in a set that are also in a set of interest (numerator), the greater the reinforcement. The greater the number of elements in

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

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**: $S_b \in E$: A (crisp) set of program elements
6: **Var**: $S_f \in E$: 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: $\quad S_b = \{y \mid (l, x, y) \in R\}$
10: $\quad$ **for all** $s \in S_b$ **do**
11: $\quad\quad$ **if** $s \notin \bar{I}$ **then**
12: $\quad\quad\quad S_f = \{y \mid (l^\top, s, y) \in R\}$
13: $\quad\quad\quad \bar{Z} = \bar{Z} \cup \{(s, (\mu_{\bar{I}}(x) \cdot \textit{deg}(1, S_b, \bar{I}) \cdot \textit{deg}(0, S_f, \bar{I}))^\alpha)\}$
14: $\quad\quad$ **end if**
15: $\quad$ **end for**
16: **end for**
17: **return** $\bar{Z}$

Figure 3: Function *analyzeRelation*

a set (the denominator), the smaller the specificity. An additional unit is added when using the range of the direct relation (line 9), since this range may be completely disjoint from the set of interest. This case cannot occur with the transpose relation (line 12) since, by definition, at least one element (x) will be part of the set of interest $\bar{I}$.

$$\textit{deg}(t, U, \bar{V}) = \frac{t + |U \cap \bar{V}|}{|U|} \quad (2)$$

3.4 Example

Our algorithm can best be illustrated through an example. We use an example from the JHotDraw drawing application (version 5.4 beta 2).[5]

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 $\alpha = 0.25$.

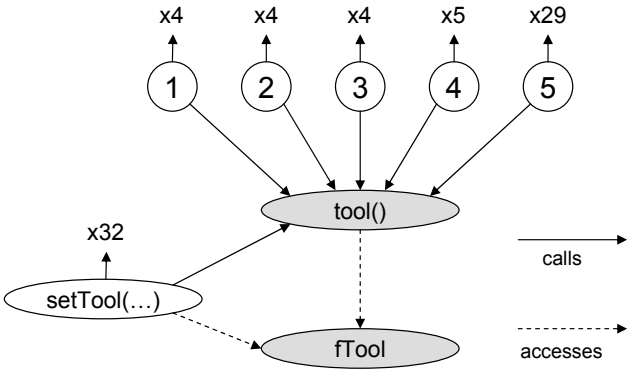

Figure 4: Partial dependencies in JHotDraw

The execution of *analyzeRelation* with $l =$ *called by* (line 9 of Figure 3) yields six callers, represented in Figure 4 by the five numbered nodes and the `setTool(...)` node. In the figure, elements in the set of interest $\bar{I}$ are shaded gray. Since none of the six nodes is in $\bar{I}$, their direct degree is $1/6$ (equation 2). For each of the six methods in the range, 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 33

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

methods: `tool()` and 32 other ones (names not important). Since `fTool` is a field and not called by any method, a first suggestion set (line 6 of Figure 2) can already be generated for the *called by* relation:

method	1	2	3	4	5	`setTool(...)`
degree	0.43	0.43	0.43	0.41	0.27	0.27

At this point the degree for `setTool(...)` as calculated using the *called by* relation is relatively low. This value is calculated by multiplying $1/6$ (direct relation) by $1/33$ (transpose relation) and taking the result to the power of 0.25. The value is low mostly because `setTool(...)` has low specificity with respect to the `calls` relation (it calls 32 methods besides `tool()`).

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

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

The final iteration analyses the relation *accessed by* and considerably changes the suggestion set. The method `tool()` is of course not accessed by anything so the analyses focuses on `fTool`. As shown in Figure 4, `fTool` is only accessed by 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. The final suggestion set is:

method	1	2	3	4	5	`setTool(...)`
degree	0.43	0.43	0.43	0.41	0.27	1.00

This result has a meaningful application since `setTool` is a non-trivial 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 case, the name of the method is a good indication of its relevance to the set of interest. However, the strength of our technique is that the same result would have been obtained even if the method had not been appropriately named. Furthermore, 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.

3.5 Complexity

The space complexity of our suggestion algorithm is negligible as it only needs temporary storage for small subsets of an entire program.[6] 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 to a program element is a small constant, the execution time of the algorithm can be modeled as $O(|L| \times |\bar{I}|)$.

[6]The static analysis required to execute the algorithm is discussed in Section 4.1.

4. CURRENT IMPLEMENTATION

We built a prototype implementation of our proposed algorithm to analyze Java programs using the four relations: *calls*, *called by*, *accesses*, and *accessed by*. Our current implementation of the algorithm relies on an in-memory program database storing all the relation tuples $(l, x, y) \in R$.

4.1 Static Analysis

The program database is built by parsing all source code files in a software system, detecting relations between different elements, and inserting each relation and its transpose in the database. The time complexity of this phase of the algorithm is linear in the total number of lines of source code in the program, and the space complexity is linear in the total number of relations recorded (also a direct factor of the total size of the program).

To ensure that the results would be as useful as possible, we have implemented the function returning the range of a relation (lines 9 and 12 of Figure 3) to return only the elements defined in the source code analyzed (as opposed to binary libraries). 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.

Another important consideration when designing the static analysis used to build the program database was the specification of the semantics of the *calls* relation 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 play out in practice, we implemented our prototype with the two alternative semantics for the *calls* and *called by* relations: to include only static bindings, and to include all methods potentially called as generated using class hierarchy analysis (CHA).

4.2 Tool Support

We implemented our algorithm as an Eclipse plug-in. Eclipse is an integrated software development environment supporting the addition of functionality [14]. Our plug-in performs the static analysis by parsing the Java files that are part of an Eclipse project. The graphical user interface support for using our algorithm was designed to be as simple to use as possible. Our implementation thus consists of a single tree view. Developers can create and name boxes representing sets of interest (the roots of the trees), and drag and drop Java elements from Eclipse views into each box. It is possible to adjust the membership degree of each element in a set of interest through a slider bar, and to filter out all elements with a membership degree lower than a user-specified threshold. Clicking a button in the main Eclipse tool bar (not shown) applies the algorithm to a selected set of interest, which automatically gets extended with the suggestions set. Figure 5 shows a view of our plug-in.

5. QUANTITATIVE EVALUATION

As part of the evaluation of our technique, we were interested in studying how the algorithm would behave in realistic conditions. Understanding the basic behavior of the algorithm was our first step in assessing the general usability of the technique. A qualitative evaluation of the usefulness of the approach in the context of a program investigation task is presented in the next section.

For our quantitative evaluation, we were specifically interested in determining:

- The typical size of suggestion sets produced for a singleton set of interest.
- The impact of the parameter α on the degree of the elements in the suggestion set, when using both static signatures and CHA to determine the *calls* relation.
- The stability of the relative order of membership degree for the elements of a suggestion set with respect to varying α and semantics for the *calls* relation.

An analysis of each of these factors helped us better configure the parameters of the algorithm and understand its output. To study the factors mentioned above, we ran different configurations of the algorithm on a series of sets of interest consisting of a single element. Although singleton sets of interest generally do not exercise the reinforcement aspect of the algorithm (except in the case of two-element cycles), they exhibit important properties that make them a very good baseline for studying the behavior of the algorithm. These properties are discussed below, where applicable.

As targets for this experiment, we used two open-source systems designed for different application domains: the JHotDraw drawing program mentioned in Section 3.4 and the Azureus BitTorrent client. [7] Both of these systems are developed in Java and Table 1 lists their characteristics as computed by the LOCC analysis program version 3.3.[8]

To derive sets of interest on which to apply our algorithm, we first selected the five classes in each system that were the most modified (i.e., whose corresponding file had the highest revision number in the CVS repository). We chose these classes because we hypothesize that heavily modified program segments are likely to be modified again, and thus form natural targets for our technique.

[7] http://http://azureus.sourceforge.net/

[8] http://csdl.ics.hawaii.edu/Tools/LOCC/LOCC.html

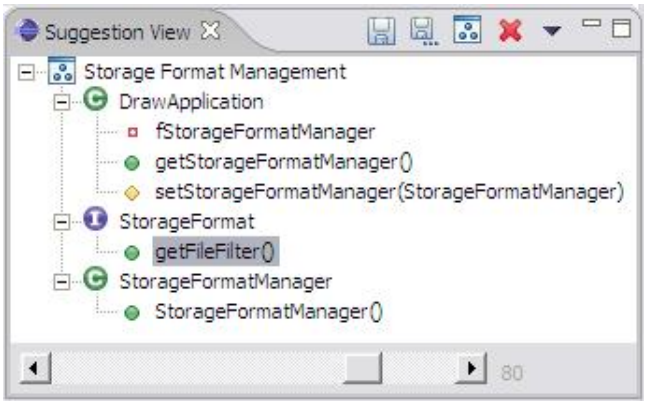

Figure 5: Graphical user interface for our suggestion algorithm

Table 1: Characteristics of Target Systems

System	# Types	# Methods	LOC
JHotDraw version 5.4 beta 2	302	2682	20,985
Azureus version 2.2	1415	7948	128,214

Table 2: Classes Analyzed

Class	Revision	# Members
DrawApplet	14	57
DrawApplication	31	110
JavaDrawApp	18	19
StandardDrawingView	30	114
TextFigure	19	61
ConfigView	181	26
DiskManagerImpl	193	77
MyTorrentsView	176	73
PEPeerControlImpl	207	165
TRTrackerClientClassicImpl	149	105

We then created singleton sets of interests with each element (field or method) in each of the 10 classes chosen. Not surprisingly, most of the 10 classes selected with our criterion were complex classes with a large number of elements. In total, we obtained 807 different singleton sets of interest. We then applied our algorithm to each set for an α value varying between 0.1 and 0.9 in 0.2 increments, and using both CHA and static bindings for the *calls* relation and its transpose.

Table 2 lists each class analyzed, its revision number, and the number of elements it declares. The top five classes in the table are from JHotDraw and the bottom five from Azureus.

5.1 Size of Suggestion Sets

The first question we investigate is *what is the typical size of a suggestion set?* The size of a suggestion set is the number of elements in relation with the elements in the set of interest. This number is independent from the value of the α parameter. In this case it is useful to analyze singleton sets of interest since suggestion sets for more than one element will be bounded by a multiple of this value corresponding to the number of elements in the set. In other words, the size of a suggestion set for a set of interest of two elements will be at most the sum of the size of the suggestion sets corresponding to the sets of interest for each individual element. Table 3 shows the average and maximum size of suggestion sets for elements in each of the 10 classes (the minimum is 0 or 1 in most cases). In the table, the column labels with suffix "-S" indicate results using static bindings and the column labels with suffix "-C" indicate results using CHA. We make three principal observations from this data.

- *On average* users of the algorithm can expect relatively small suggestion sets for a singleton set of interest. Applying the algorithm without CHA over the 807 test sets yields an average suggestion set size of 4.8, with per-class averages in the interval [3.6–6.1].
- *A small number of elements* yield much larger suggestion sets. The maximum values calculated help us determine the worst case scenario. For example, the largest suggestion set produced with our analysis contains 80 elements. A suggestion set generated from two elements with non-overlapping suggestion sets can thus yield a suggestion set of 160 elements. In such cases, it is unrealistic for developers to look

Table 3: Size of suggestion sets

Class	Avg-S	Max-S	Avg-C	Max-C
DrawApplet	3.9	19	5.7	30
DrawApplication	3.9	29	5.5	46
JavaDrawApp	4.4	33	5.4	34
StandardDrawingView	3.6	22	8.4	70
TextFigure	4.3	19	7.6	53
ConfigView	4.7	31	9.9	80
DiskManagerImpl	5.6	51	8.3	64
MyTorrentsView	4.6	23	5.9	24
PEPeerControlImpl	6.1	31	8.8	86
TRTrackerClientClassicImpl	5.6	46	6.4	51
Aggregated Values	**4.8**	**51**	**7.3**	**86**

at all of the elements suggested. It is exactly for this reason that we have designed our algorithm to produce fuzzy sets: developers can automatically filter out elements with a low degree from the suggestion set.

- *Applying the algorithm with CHA* invariably generates larger suggestion sets. This result is not surprising since using CHA can only add dependencies. The empirical data documents the extent of the phenomenon.

5.2 Parameterization of the Algorithm

Using the algorithm in practice requires choosing a value for the α parameter. To assess how the parameter value impacts the results of the algorithm, we analyzed the data produced by running our algorithm on our 807 test sets under the 10 different configurations mentioned earlier. For each suggestion set produced, we counted the number of elements with degree above 0.5. This strategy was designed to produce values that would indicate what a developer would see as suggestions when using a filtering threshold of 0.5. To give a sense of the variability of the results, we average over each class. Figure 6 shows the results of this analysis. Each bar in the graph represents the ratio of elements with degree above 0.50 for a specific class under a specific configuration of the algorithm. We tested the 10 configurations (five values of α with both static bindings and CHA), for all elements in all 10 classes. White columns represent JHotDraw classes and grey columns represent Azureus classes. The suffix "-S" denotes configurations using static binding while "-C" denotes use of CHA to calculate the *calls* relation.

We make four observations based on this data.

- For a given semantics of the *calls* relation, ratios are exactly the same for $\alpha = 0.9$ and $\alpha = 0.7$.
- For values of $\alpha >= 0.7$ many suggestion sets contain no or very few elements above 0.5.
- For values of $\alpha = 0.1$ almost all of the elements in the suggestion set have a degree above 0.5.
- There is significant variability in the ratio of elements above 0.5, both between classes and between applications.

The first three observations synthesize the exponential impact of the α parameter on degree values. Given that α is used as a exponent in the unit interval it is expected that its impact will not be felt beyond a certain threshold. Our experiment documents this threshold as measured with 807 suggestion sets of varying size. Variations observed between different classes can partially be explained

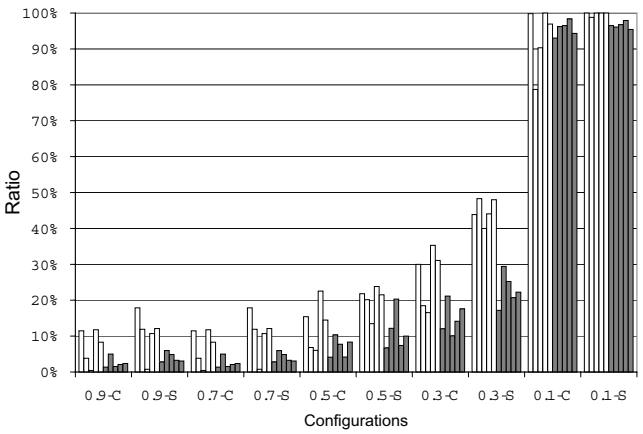

Figure 6: Per-class ratio of elements with degree above 0.5 in the suggestion sets

by the different roles that classes play in class hierarchies. As an example, we can contrast two classes from the JHotDraw system under the $\alpha = 0.3$ configuration with CHA. The class generating the highest ratio of high-degree elements is `JavaDrawApp` (35%), while the class with the lowest ratio is `TextFigure` (17%). Inspection of the program shows that `TextFigure` is part of a deep class hierarchy heavily using polymorphism, while `JavaDrawApp` is the entry point to the application. For this reason, specificity for `TextFigure` is expected to be lower, and correspondingly so is the number of high-degree elements.

These observations provide us with two valuable results. First, the most useful value of α is in the interval $0.1 < \alpha < 0.7$. Second, the variability observed between classes indicates that we should not treat degree values as an absolute measure of how interesting an element is, but rather as a relative measure in the context of a suggestion set.

5.3 Stability of the Algorithm

The last observation described in the previous section indicates that the relative order of elements in a suggestion set according to their degree is more important than the absolute value of their degree in helping a developer determine potential elements of interest. Because of this result, we wanted to ensure that the order of elements was stable. We thus recorded the element with the highest degree in the suggestion set produced for each set of interest for each configuration and analyzed whether this element was the same across all 10 configurations. Table 4 shows the results for each class and the aggregated results for all 807 sets of interest. In the table, the column labeled *Ratio* shows the ratio of suggestion sets whose element with highest degree does not change across all configurations to all of the sets generated from elements of this class. The column labeled *Var* shows the maximum number of elements with highest degree. For example, for 95% of the sets generated from elements of class `DrawApplet` the element with the highest degree in the suggestion set is the same regardless of the algorithm's parameters. For the three remaining sets, the element with highest degree varies only between two alternatives. Overall, 86% of the 807 sets tested have the same element of highest degree independent of the configuration chosen. This data indicates that the algorithm is generally stable in its recommendation of the most interesting element.

Table 4: Ratio of stable sets

Class	Ratio	Var
DrawApplet	95%	2
DrawApplication	90%	3
JavaDrawApp	84%	3
StandardDrawingView	82%	2
TextFigure	77%	3
ConfigView	85%	2
DiskManagerImpl	90%	2
MyTorrentsView	82%	2
PEPeerControlImpl	80%	2
TRTrackerClientClassicImpl	93%	2
Aggregated Results	**86%**	**3**

6. CASE STUDIES

In the previous section, we quantitatively assessed the nature of the results generated by our algorithm. Although this analysis provides us with useful insights into the behavior of the algorithm under realistic conditions, it does not provide evidence that the algorithm is useful to software developers. This section presents 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 methods of interest to developers investigating source code. We selected our case studies by identifying, in each target system, a high-level concern that was partially implemented by one of the five most modified classes (i.e., the classes identified in the previous section).

6.1 JHotDraw Study

Our first case study discusses a scenario of program investigation involving class `TextFigure`. This class represents a box where users of JHotDraw can enter and edit text. In the version of JHotDraw used, `TextFigure` supports three attribute types: font size, font style, and font name (in addition to general attributes supported by a super class). Let us assume that a developer is asked to enhance the class to support additional attribute types. This is not a trivial task as the mechanism for attaching attributes to figures is much more elaborate than simply adding a key-value pair to a properties object. For example, in class `TextFigure` there are two methods named `setAttribute`, one of which is 24 lines of source code long.

We used our Eclipse plug-in and built a set of interest consisting of the two methods of `TextFigure` named `getAttribute` and the two methods named `setAttribute`. This is a realistic starting point for a developer unfamiliar with the code of JHotDraw since it consists of all of the members of `TextFigure` containing the text string "atttribute".

Applying our algorithm to this set with $\alpha = 0.3$ and CHA yielded the suggestion set of Table 5.

This suggestion set is small enough to be completely investigated. Going through the list, we selected each member likely to be helpful in understanding the mechanism for setting attributes on a `TextFigure` (in bold in the list). Our justification is as follows. Elements 1 and 5 are selected because they help understand the mechanism used to create and manage constants attached to properties. Elements 3, 4, and 8 are selected because they are the constants referring to the properties of a `TextFigure`. Elements 6 and 10 are selected because they are the getter and setter methods for a font object. Finally, element 7 is added because it contains a call to generate default font properties. Elements 2 and 9 are rejected because they only access the properties and provide no insight into the mechanism. Elements 13 and 14 are rejected because they deal with general attributes not specific to `TextFigure`, and all other elements are rejected because they are simply wrappers forwarding calls to `setAttribute` and `getAttribute`.

Table 5: Suggestion set for JHotDraw: first iteration

#	Element	Degree
1	**FigureAttributeConstant.getConstant(...)**	**0.62**
2	AttributeFigure.writeObject(...)	0.60
3	**FigureAttributeConstant.FONT_SIZE**	**0.55**
4	**FigureAttributeConstant.FONT_STYLE**	**0.55**
5	**FigureAttributeConstant.equals(...)**	**0.55**
6	**TextFigure.getFont()**	**0.52**
7	**TextFigure.TextFigure()**	**0.52**
8	**FigureAttributeConstant.FONT_NAME**	**0.49**
9	TextFigure.drawFrame(...)	0.42
10	**TextFigure.setFont(...)**	**0.41**
11	DecoratorFigure.getAttribute(...)	0.40
12	GraphicalCompositeFigure.getAttribute(...)	0.40
13	AtttributeFigure.setAttribute(...)	0.39
14	AttributeFigure.getAttribute(...)	0.37
15	DecoratorFigure.setAttribute(...)	0.36
16	GraphicalCompositeFigure(...).setAttribute(...)	0.36
17	GroupFigure.setAttribute(...)	0.32

Table 6: Filtered suggestion set for JHotDraw: second iteration

#	Element	Degree
1	TextFigure.createCurrentFont(...)	0.89
2	DrawApplication.createFontStyleMenu()	0.86
3	DrawApplication.createFontSizeMenu()	0.86
4	TextFigure.fFont	0.78
5	DrawApplication.createFontMenu()	0.78
6	DrawApplet.createFontChoice()	0.78
7	ColorMap.color(...)	0.72
8	FigureAttributeConstant.FigureAttributeConstant(...)	0.72
9	FigureAttributeConstant.getConstant(...)	0.72
10	AttributeFigure.initializeAttributes()	0.64
11	AttributeFigure.writeObject(...)	0.64
12	FigureAttributeConstant.getName()	0.62

After this first iteration, we created a second set of interest by adding all the elements selected in the suggestion set to the initial set of interest and raising their degree to 1.0. We then applied the algorithm to this second set of interest to generate a broader suggestion set. The suggestion set generated in this way contained 58 new elements scattered in 25 different classes. In such a case, the possibility of filtering off elements based on their membership degree becomes a necessity. Since there were 28 elements in the suggestion set with a membership degree equal to or above 0.5, we filtered with the more restricting value of 0.6 instead, yielding the following suggestion set of 12 elements shown in Table 6.

Among these 12 elements, only two (7 and 11) are not relevant. Most other elements can be considered very relevant. In particular, element 1 creates the default font used for new text figures, and elements 2, 3, and 5 create the menus that allow users to modify the font attributes of text figures. A developer wishing to add a new attribute would most likely have to understand these methods. Within the elements with membership degree lower than 0.60, a detailed inspection finds that only two other elements would also clearly be useful to investigate. These elements are listed in Table 7 with their order in the sorted list of members, and their memberships degree. The complete suggestion set produced is available on our website. [9]

This case study illustrates that our algorithm can be used to quickly identify a core set of elements of interest. Because the algorithm only selects direct dependencies to elements in the input

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

Table 7: Other relevant elements in the second suggestion set for JHotDraw

#	Element	Degree
15	FigureAttributeConstant.getID()	0.58
35	DrawApplet.setupAttributes()	0.42

set, it is clear that a single iteration does not produce the *complete* set of elements of interest to a developer. However, applying the algorithm for a small number of iterations can mitigate the painstaking manual inspection of dependencies needed to build a core set of elements to investigate. Since there is evidence that a a good starting point can lead to more productive program investigation activities [17], techniques that can inexpensively provide this starting set have to potential to lead to a significant improvement in the efficiency of program investigation activities.

6.2 Azureus Study

Our first case study provides basic evidence of the usefulness of the algorithm, but is subject to investigator bias. We performed a second 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 system. 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`. As described in Section 5, this 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 applied our algorithm on this set of interest with CHA and $\alpha = 0.3$. The resulting suggestion set comprised 54 elements, which we merged with the set of interest to produce a set of fields and methods of potential interest to a developer wishing to understand the implementation of the file allocation concern.

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,[10] the SA4J static analysis tool,[11] the JProbe profiler,[12] and manual analysis of the source code.

The experts were asked to look at each element in the merged 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 also asked to justify their decisions. The experts were unaware of the reason why 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.

Working as a team for over one hour and using the features of Eclipse, the experts produced a qualification of each element in the suggested set. Out of 58 elements in the list, 31 (53%) were marked as relevant, 12 (21%) were marked as somewhat relevant, and 15 (26%) were marked as not relevant. Since 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. 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. Figure 7 shows the results. In the figure, the middle bar represent the ratio over the entire set of 58 elements. Bars to the left represent intervals of high-degree elements and bars to the right intervals of low-degree elements. From this figure, we clearly see that, overall, elements marked by our algorithm with a high degree were also relevant to developers. For example, among the 10 elements with highest degree (including the 4 elements with degree 100 used as set of interest), 9 are marked as either relevant or somewhat relevant. On the other hand, of the bottom 10 elements only 5 are marked as relevant or somewhat relevant. In general, for all top intervals (as calculated with our algorithm), the ratio of relevant elements (as determined by the experts) is above average, and vice-versa for bottom intervals. In brief, the Azureus study documents a realistic case of a program investigation task where high-degree elements produced by our algorithm corresponded to elements of interest for a developer performing the task.

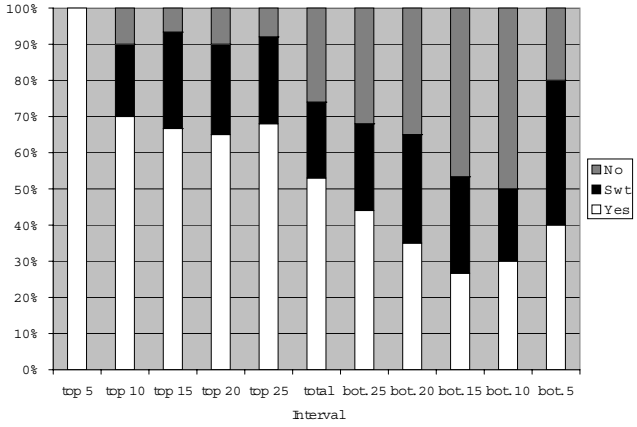

Figure 7: Ratio of relevant elements for different intervals

7. DISCUSSION

7.1 Experimental Critique

The observations of Section 5 are influenced by our choice of sample sets. These sample sets are come from very specific sources: members of heavily modified classes in two specific systems. For these reasons, the quantitative evaluation of our algorithm may not be representative of the average use by a developer. However, we surmise that our general observations will generalize since they are based on results that are consistent over a population of suggestion sets that greatly vary in size (see Table 3).

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.

The interpretation of the JHotDraw 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, and the detailed justification for the relevance assessment is included in the paper. This level of detail allows independent researchers to understand our results in the light of their own interpretation. The evidence provided by the JHotDraw 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. How-

[10] http://www.cs.ubc.ca/labs/spl/projects/feat
[11] http://www.alphaworks.ibm.com/tech/sa4j
[12] http://www.quest.com/jprobe/index.asp

ever, 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, so that it can be assessed independently. The case studies thus provide reliable evidence that the technique can be useful in realistic conditions.

Our studies also did not assess the number of relevant elements *not* uncovered by the technique. Given the early development stage of our technique and the difficulty of obtaining a reliable benchmark of all of the code of a system relevant to a concept, we postponed the evaluation of recall to future work.

Finally, at this stage of the research it is also difficult to determine how well the results will generalize to different types of software modification tasks. Although our initial experience is encouraging, more elaborate empirical testing will be required to help us answer this question.

7.2 Additional Applications of the Technique

An important number of techniques in software engineering rely on the creation and maintenance of links between source code and other artifacts. Examples include many of the techniques discussed in Section 2, but also approaches to model the implementation of concerns in source code [16] and to automatically associate source code with the elements of a UML model [6]. The basic question many of these approaches try to answer is: *given a concept, concern, feature, model, or other high-level construct, what source code corresponds to this construct?* Unfortunately, the answer to this question is very seldom crisp. For example, if a feature uses a general library function m, should m be associated with the feature? While including m may be useful in certain cases, in other contexts it may not. Our proposed technique can be used to alleviate the problem of deciding which elements to include in a concept mapping structure by supporting fuzzy boundaries. In other words, given a crisp set describing program elements associated with a high-level concept, applying our algorithm is equivalent to automatically extending the boundaries of the set to include elements directly related (according to specific static dependencies) to the elements in the set, but whose degree of association with the set is weighted according to our criteria of specificity and reinforcement.

8. CONCLUSION

In this paper, we presented a technique to automatically suggest elements of potential interest to a developer involved in a program investigation task. Our technique takes as input a 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 to 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 that our algorithm can help developers quickly select program elements worthy of investigation while avoiding less interesting ones.

Acknowledgments

The author is thankful to Félix Martineau and Philippe Nguyen for sharing their knowledge of the Azureus system and to Davor Čubranić, Nomair Naeem, Rob Walker, the members of the Software Practices Lab at UBC, and the anonymous referees for their valuable comments on this paper. This work was supported by a McGill University start-up package and by NSERC.

9. REFERENCES

[1] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 246–256, 1990.

[2] G. Antoniol, G. Canfora, A. De Lucia, and E. Merlo. Recovering code to documentation links in OO systems. In *Proceedings of the 6th Working Conference on Reverse Engineering*, pages 136–144, 1999.

[3] T. J. Biggerstaff, B. G. Mitbander, and D. E. Webster. Program understanding and the concept assignment problem. *Communications of the ACM*, 37(5):72–82, 1994.

[4] B. W. Boehm. Software engineering. *IEEE Transactions on Computers*, 12(25):1226–1242, 1976.

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

[6] A. Egyed. Resolving uncertainties during trace analysis. In *Proceedings of the 12 International Symposium on the Foundations of Software Engineering*, pages 3–12, 2004.

[7] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *IEEE Transactions on Software Engineering*, 29(3):210–224, 2003.

[8] K. B. Gallagher. Visual impact analysis. In *Proceedings of the International Conference on Software Maintenance*, pages 52–58, 1996.

[9] K. B. Gallagher and J. R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8), 1991.

[10] T. Gyimóthy, Árpád Beszédes, and I. Forgács. 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*, pages 303–321, 1999.

[11] D. Jackson and E. J. Rollins. A new model of program dependence for reverse engineering. In *Proceedings of the 2rd ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 2–10, 1994.

[12] M. Kersten and G. C. Murphy. Mylar: a degree-of-interest model for IDEs. In *Proceedings of the 4th Conference on Aspect-Oriented Software Development*, pages 159–168, 2005.

[13] P. Kruchten. *The Rational Unified Process: An Introduction.* Addison-Wesley, 2nd edition, 2000.

[14] Object Technology International, Inc. Eclipse platform technical overview. White Paper, 2001.

[15] M. P. Robillard, W. Coelho, and G. C. Murphy. How effective developers investigate source code: An exploratory study. *IEEE Transactions on Software Engineering*, 30(12):889–903, 2004.

[16] M. P. Robillard and G. C. Murphy. Concern Graphs: Finding and describing concerns using structural program dependencies. In *Proceedings of the 24th International Conference on Software Engineering*, 2002.

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

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

[19] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.

[20] P. Tonella, G. Antoniol, R. Fiutem, and E. Merlo. Variable precision reaching definitions analysis for software maintenance. In *Proceedings of the 1st Euromicro Conference on Software Maintenance and Reengineering*, pages 60–67, 1997.

[21] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

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

[23] A. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Transactions on Software Engineering*, 30(9):574–586, 2004.

[24] W. Zhao, L. Zhang, Y. Liu, J. Sun, and F. Yang. SNIAFL: Towards a static non-interactive approach to feature location. In *Proceedings of the 26th International Conference on Software Engineering*, 2004.

[25] H.-J. Zimmermann. *Fuzzy Set Theory and Its Applications.* Kluwer Academic Publishers, third edition, 1996.

[26] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *Proceedings of the 26th International Conference on Software Engineering*, pages 563–572, 2004.