IBM CASCON Conference
Toronto, November 10-13, 1997

Similarity-Based Retrieval for Diverse Bookshelf Software Repository Users

Igor Jurisica
Department of Computer Science, University of Toronto
Toronto, Ontario M5S 1A4, Canada, juris@ai.utoronto.ca

Abstract

The paper presents a similarity-based retrieval framework for a software repository that aids the process of maintaining, understanding, and migrating legacy software systems [12].

Designing a software repository involves three issues: (1) information content; (2) information representation; and (3) strategies for accessing repository artifacts. Assuming the architecture presented in [12] we extend the retrieval system to support imprecise queries, iterative browsing, and diverse users. Because of repository size, complexity of queries and relations among artifacts, we take a performance approach to support a scalable implementation.

We propose a retrieval system that uses numeric [31] and semantically rich context-based similarity [19]. Efficient iterative browsing is based on an incremental query evaluation algorithm from database management systems [20]. Explicitly defined context supports various retrieval strategies and diverse user models.

1. Introduction

Software repositories are used to store and retrieve information about software artifacts. The repository must support different tasks and diverse users efficiently. Thus, retrieval strategies cannot be predefined, and the system should support imprecise queries and iterative browsing.

Diverse techniques are used to retrieve software artifacts from a repository. They include library cataloging, database classification, query-based retrieval [42], formal specification [37], information retrieval [7], hypertext-based classification and retrieval [17], similarity-based organization [25, 39], clustering and browsing [5, 34], case- [11, 13] and analogy-based retrieval [30], and hybrid organizations [4].

Similarity-based retrieval helps to access relevant artifacts when the query is not complete and precise by iteratively relaxing or restricting the initial query [15, 34]. This becomes useful when a repository is large, and created by many users over a longer time period. Generally, it is impossible to decide in advance what needs to be retrieved, when, and in what form. Flexible and iterative similarity-based retrieval diminishes this problem [26].

CAReT is an analogy-based retrieval system applied to software design reuse [30]. It supports incomplete queries, and similarity-based retrieval of software artifacts. Background knowledge and the component design library comprise a knowledge base. The system uses *is-a* and *part-of* links to navigate through the library. The retrieval mechanism uses a three-phase procedure: locate, evaluate and select. The system uses two levels of matching: object- and attribute-based. The first level compares object classes, object types, attribute properties, and their relationships to other object types. The second level compares attribute data types and properties.

AIRS is an AI-based library system for software reuse [36]. It integrates faceted indexing and a hierarchical frame system. Similarity assessment between two components uses their descriptions, and is based either on the closeness or the subsumption relation. The closeness relation captures the fact that new components can be constructed by modifying existing constructs. The subsumption relation captures the idea that certain components can be built by combining several other components together. Based on this measure, two components are similar if the overall difference between their corresponding terms is small. Because the components must be described in terms of all features, adding new features involves modifying all components that are already stored in the library.

Constantopoulos et al. [4] presents a conceptual knowledge base, called SIB, to support software reuse. The *Telos* language [33] is used to represent requirements, design and code. SIB's query system is based on graphical traversal of semantic links.

The use of similarity-based retrieval in databases is described in [22]. Termsinsuwan et al. [43] describes how case-based reasoning facilitates reuse of abstract data types. Henninger [18] introduces a case-based decision system for knowledge reuse in software engineering by means of formal domain analysis.

Finnigan et al. [12] presents a software repository, called Bookshelf, which was designed and developed to help in capturing and analyzing relevant information about legacy systems. Software artifacts are represented using *Telos* [33]. A developer accesses software artifacts in the repository by browsing its content. The repository architecture also includes a set of tools that are used to populate the repository and to reverse-engineer software artifacts.

Generally, one retrieval strategy is not sufficient to cover diverse tasks. In general, two types of queries may be submitted: exact and approximate. The user may specify an initial query directly, or using query-by-example. When the answer does not satisfy quantity or quality requirements, query-by-reformulation is initiated. Artifacts similar to the retrieved ones can be explored by modifying the initial query [1, 24].

This paper describes a similarity-based retrieval system for a software repository that represents information about legacy software systems. We assume the architecture presented in [12], and we focus on retrieving all (and only) relevant repository items even for an imprecise query. Our objective is a retrieval system that is flexible enough to support diverse user models. Users have different expectations on Bookshelf, look for diverse information, and use different search patterns. Because of repository size, complexity of represented relations and query complexity, the system must be scalable.

To achieve this functionality, we adopt an efficient context-based retrieval algorithm from the case-based reasoning system ***TA3*** [19, 20, 21]. The algorithm supports imprecise queries, and automatic query transformations that control quantity and relevance of retrieved artifacts. The system's flexibility is accomplished by explicitly stating context in the similarity definition, and by using the *Telos* [33] representation language.

2. Retrieval in Bookshelf

We have identified tasks pertinent to novice and expert software engineers within a specific project. A novice user's main task is familiarization. The software repository should assist novices in fast and in-depth learning. A hierarchical access to repository artifacts and imprecise queries must be supported.

Expert users have domain knowledge, and their retrieval strategies are task-dependent. *Redocumentation and design recovery* facilitates understanding a system, design decision taken, and the logic behind the implementation. *System migration* supports translating a system from one language to another, changing software architecture, or porting a system to another platform. *Maintenance and evolution* covers bug fixing and functionality improvement of a system. *System testing* assists correctness and performance analysis. *Requirements' reengineering helps* in extracting, analyzing, understanding and modifying system requirements.

Generally, software artifacts can be accessed by selecting them in diagrams, by specifying a query (a search-based approach), by specifying a sample artifact (a query-by-example approach), and by modifying the earlier query (a query-by-reformulation approach). These approaches are further extended to include similarity-based retrieval. The next section describes context-based similarity, and query transformations.

2.1. Similarity-Based Retrieval

Traditionally, information retrieval has been concerned with representation of texts and comparison of these representations. Software engineering research contemplated abstracting source code to make the retrieval process easier by providing extra information [2, 10, 43]. In addition, users rarely specify query precisely and/or completely. As a result, uncertain and imprecise information must be taken into account when designing retrieval systems. Numerous approaches support imprecise queries, most of which are based on fuzzy logic [38, 45].

It is expected that a software repository supports diverse information retrieval problems. Thus, different interaction methods and diverse information-seeking strategies must be available. Retrieval strategies may change not only between individual sessions but even within a particular one. Users rarely specify their information needs, mainly because conception of information problems may change in the process. More recently it was observed that information retrieval is an inherently interactive process [1, 24]. Thus, the retrieval system should deploy an efficient iterative browsing [5, 34], which provides the means for exploring the "neighborhood" of a repository artifact. Such neighborhood depends on the task, current state of the search, and the repository state. This means that the neighborhood of an artifact is context-dependent.

2.1.1. Context-Based Similarity

Similarity-based retrieval supports the retrieval of exact and similar matches [22]. Relevant artifacts are retrieved even for an imprecise query. In addition, such retrieval aids in exploring the neighborhood of a given artifact [25, 19]. Explicit context in the query modifies the search pattern and similarity assessment outcome [26].

It is assumed that *repository artifacts* are represented as structured objects with relations [33]. Objects and relations are represented as a set of attributes and their values:

$$A = \{\langle a_0{:}V_0\rangle, \ldots, \langle a_n{:}V_n\rangle\},$$

where $\langle a_i{:}V_i\rangle$ is an attribute-value pair. Each attribute a_k has an associated attribute domain D_k. All values for a given attribute must belong to its domain. Using information about usefulness and properties,[1] attributes are grouped into one or more *Telos categories* [33]. Categories bring additional structure to information representation. This decreases the impact of irrelevant attributes on system performance [19, 35].

For example, dsinit is a function – a repository artifact with the following attributes:

⟨name: dsinit,
formal arguments: (int, float),
return type: float,
source location: xx_dsini⟩.

[1] This information comes from either domain knowledge or via knowledge mining.

These attributes specify a function signature [27]. Since not all attributes are equally important for retrieval, they should be placed into different categories. The semantics of these attributes suggest three categories: {name}, {formal arguments, return type}, and {source location}. Hence, the semantic information of attribute value types is defined through the use of context.

Context is a parameter of similarity that maps a repository onto a set of relevant artifacts. It controls partial matching by specifying attributes used during similarity assessment. A context, Ω, is defined as a finite set of attributes with associated constraints on the attribute values:

$$\Omega = \{\langle a_0{:}CV_0\rangle, \ldots, \langle a_k{:}CV_k\rangle\},$$

where a_i is an attribute name and CV_i is a constraint that specifies the set of "allowable" values for attribute a_i. A context specifies the relevant attributes and how "close" an attribute value must be to satisfy it.

We say that an object *satisfies* (or matches) a particular context (a query) if for each attribute specified in the context the value of that attribute in the artifact satisfies the constraint (i.e., the value is contained in the constraint set).

A repository artifact A *satisfies* a context Ω, denoted *sat* (A, Ω), if and only if for all pairs $\langle a_i{:}CV_i\rangle \in \Omega$, there exists a pair $\langle a_i{:}V_i\rangle \in C$ such that V_i is in CV_i:

$$\mathit{sat}\,(A, \Omega) \text{ iff } \forall\langle a_i{:}CV_i\rangle \in \Omega \rightarrow \exists V_i\, \langle a_i{:}V_i\rangle \in A \wedge V_i \in CV_i.$$

We say that the context Ω is *satisfiable* if and only if there exists at least one artifact in a repository that satisfies it:

$$\mathit{satisfiable}\,(\Omega) \text{ iff } \exists A \in \Delta\text{: } \mathit{sat}\,(A, \Omega),$$

where Δ is a repository.

Using the dsinit function mentioned above, one could specify a context as follows:

name: {dsinit},
formal arguments: {int, float},
return type: {float, int},
source location: {xx_dsini}.

Artifact A_1 and A_2 are *similar* with respect to a given context Ω if and only if both A_1 and A_2 satisfy context Ω:

$$A_1 \approx_\Omega A_2 \leftrightarrow \mathit{sat}\,(A_1, \Omega) \wedge \mathit{sat}\,(A_2, \Omega).$$

Context-based similarity is a key part of the proposed repository retrieval system. Given a repository Δ and a context Ω, the function returns a set of *relevant* artifacts $SA \subseteq \Delta$ if and only if all artifacts in *SA* are similar with respect to the given context:

$$relevant(\mathrm{SA}, \Omega, \Delta) \text{ iff } \forall \mathrm{A} \in \Delta, \mathrm{A} \in \mathrm{SA} \leftrightarrow sat(\mathrm{A}, \Omega).$$

2.1.2. Query Transformations

Query transformations control the quality and quantity of retrieved artifacts. They are executed either automatically or by the user [26]. Query-by-reformulation and iterative browsing use two transformations: *relaxation* and *restriction*.

Query relaxation extends the scope of a query and explores the neighborhood of a given artifact [26]. Relaxation is a monotonic transformation [19, 24]. It is implemented as either *reduction* or *generalization*. The former removes constraints from the context by reducing the number of attributes required to match: given *m_of_n* matching [35], the number of attributes is reduced from m to p, where $0 < p < m \le n$.[2] Generalization relaxes the query by enlarging the set of allowed values for an attribute. This is achieved by using *is-a* hierarchies or by adding values to the set of allowed values in the context.

Assume the context mentioned earlier. Relaxation by reduction results in a new context satisfiable by any function that returns float and is part of xx_dsini:

return type: {float},
source location: {xx_dsini}.

Since name, formal arguments and return type are not part of the context, they are ignored during matching. Technically, we assume that those attributes have a set of allowed values as their domains:

name: $\{D_{name}\}$,
formal arguments: $\{D_{arguments}\}$,
return type: {float},
source location: {xx_dsini}.

It should be noted that both contexts mentioned above are satisfiable by the dsinit function.

[2] Compared to *m_of_n* matching [35], additional degrees of freedom during matching is achieved by using *Telos* categories.

Restriction is analogous to relaxation, it decreases information overload on a user. *Expansion* is a transformation opposite to reduction and *specialization* is converse to generalization [21].

To support scalability and efficient query processing we implemented the transformations presented using an incremental view maintenance algorithm from database management systems. The main idea is to reuse partial results of query evaluation during iterative browsing. Small updates to the query generally produce only small changes to the query result. Thus, an incremental approach requires only local changes to the query. Efficiency improvement is more visible during iterative browsing, or when a large and/or complex repository is used [20].

2.2. Retrieval Strategies

Maarek et al. [34] define the component retrieval scenario as query specification, followed by linear retrieval and browsing. The usefulness of different information-seeking strategies depends on the user's background and the task at hand. Available strategies can be characterized on many dimensions, each with a varied combination of characteristics [1]: *Method of interaction* (e.g., scanning, searching), *goal of interaction* (e.g., learning, selecting), *mode of retrieval* (e.g., recognition, specification), and *resource considered* (e.g., information, meta-information).

Assuming different user models and active tasks, it is possible to estimate a user's goal, and then suggest an appropriate information-seeking strategy [9]. For example, a novice user is more likely to get better results using a scanning method of interaction. In contrast, a maintenance software engineer needs an efficient searching method in order to retrieve a specific repository artifact. Also, the goal of interaction for a novice user is learning, while for an expert user it is selecting.

Sixteen information-seeking strategies can be defined using the dimensions presented. Each of these strategies fits a particular user model. For example, a user searching a repository directory with a goal of selecting artifacts relevant to a particular task is characterized by the scanning method of interaction, using the recognition mode of retrieval, and searching through meta-information. A system-knowledgeable user can use meta-information to retrieve specific artifacts and thus uses the searching method of interaction. In this situation, the goal of interaction is selecting and the mode of retrieval is by specification.

2.3. The Retrieval System

The context for similarity-based retrieval can be defined by the user, inferred from the user's current actions, or automatically transformed (relaxed or restricted) by the system. When context is defined by the system, a particular user model can be used to provide information about the user's primary task, her preferences, search patterns, previous tasks, etc.[3] Recently accessed repository artifacts provide contextual information such as location (spatial similarity), data structures used, data-flow dependencies, etc.

Context-based similarity supports iterative *browsing* by *incremental query relaxation* and *restriction* [20]. Browsing is crucial in software library systems [34], and is useful in training usage scenarios, or when a repository is large and created by many users over a longer period of time. This can be used in cooperative retrieval systems, similar to the one described in [6].

Similarity-based retrieval deploys concept hierarchies of artifacts, and uses virtual links to traverse the repository and to relax or restrict the query. For example, when accessing a variable declaration, the system retrieves all modules and files that include such a declaration. In addition, similarity-based retrieval locates files and modules where this variable is fetched or stored. It also accesses variables that use the same data type. Although the system can perform query transformations automatically, the process is interactive and cooperative; the user controls the query transformation via changes to the context.

To achieve both high recall and high precision the proposed context-based similarity uses iterative browsing and features provided by *Telos*. More specifically, categories group equally important attributes together. Thus, each attribute has its own constraint and can be treated in a distinct way during relaxation and restriction. This results in a more flexible version of the *m_of_n* matching [35], which enables higher precision [19]. Incremental query revisions are computed efficiently [20].

[3] User models can be reused [9].

There are several levels of query imprecision. In addition, different representation schemes can be used to store repository artifacts. To cover the spectrum of these problems, we define the following measures used in the similarity-based retrieval module: lexical, semantic, functional, and spatial.

It is important for the proposed retrieval system (see Figure 1) to integrate with the present environment. The pre-processing module transforms a query into a format that relates to the profile of repository artifacts. Context specifies the user model, task being solved, and other information that defines retrieval goals. Criteria define a retrieval method – xxref, grep, repository retrieval, similarity-based retrieval, etc. It is assumed that context specifies default criteria for a specific user group, and for a particular task.

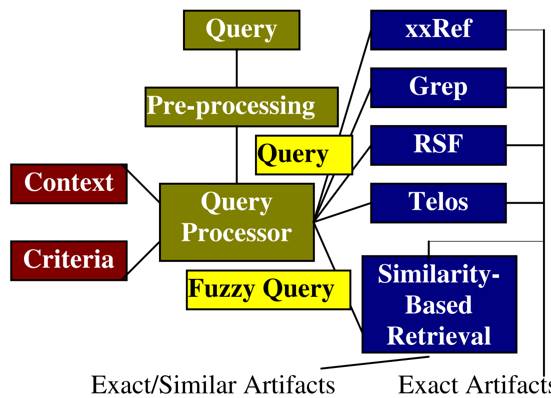

Figure 1. Retrieval system architecture

As suggested in Figure 1, the similarity-based retrieval algorithm can be used directly (i.e., for approximate queries, using lexical similarity), or iteratively (i.e., to retrieve relevant artifacts using semantic or spatial similarity).

User studies have revealed specific patterns followed by experts during information retrieval: usually, experts search for a variable store first and then for fetches [32]. This is supported by automatically inferring context from the recently accessed artifact.

2.3.1. Lexical Similarity

Lexical similarity includes textual similarity, in particular name matching. There are two possible approaches: a *contrast model* [44] and *n-gram indexing*. For the contrast model, assume $|P_Q|$ represents the length of the query pattern and

$|P_i|$ represents the length of the repository artifact pattern. We define $|P_Q \wedge P_i|$ as the length of the pattern that is present in both P_Q and P_i. Then the lexical similarity is defined as follows:

$$\frac{\left|P_Q \wedge P_i\right|}{\left|P_Q\right| + \left|P_i\right| - \left|P_Q \wedge P_i\right|}$$

Lexical similarity only deals with lexical problems. The contrast model or *n*-gram indexing would not find xx_string and xx_text to be similar.

If we consider query xx_init and files: xx_intg, xx_dsini, xx_string and xx_point from one package, contrast model would select xx_intg and xx_point as the closet matches. A simple *3*-gram would select xx_intg as the closest match, followed by xx_dsini and xx_point. However, taking naming conventions into account, a simple heuristic can be used to select xx_dsinit as the closet match to xx_init. This selection is coherent with the semantic similarity of respective files.[4]

2.3.2. Semantic Similarity

The problem of lexical similarity that arises with the example of xx_string and xx_text can be diminished by using background knowledge and *part-of* relationships. Since string is part of the text, xx_string and xx_text are similar with respect to *part-of* relationship. This, however, requires a semantically rich lexicon. Currently, this kind of matching is implemented by having the user state such relations in context.

Semantic similarity also enables us to find variables that use similar data structures as returned value or a function parameter. Distance between artifacts is determined using i*s-a* and *part-of* hierarchies of data types. This approach is similar to [43] but context-based similarity controls, which features of the data type description are more important for a specific search. This distinction is crucial for both high recall and precision. Next we describe how semantic and functional similarities are related.

2.3.3. Functional Similarity

Functional similarity covers similarity according to functional description of the artifact. Methods such as described in [10] can be used. Our system can use functional similarity once the abstracted description is provided. The similarity is determined between a functional description in the query and functional descriptions found in the repository, taking into account relevance of individual features. The comparison is achieved using pattern matching on function signatures [27]. For example, the following signature specifies functions of any name, integer return type, parameters of type integer, float and bit value, and that can be in any module:

⟨name: D_{name},
formal arguments: (int, float, bit value),
return type: integer,
source location: $D_{location}$⟩.

The matching process can be tuned up by using information about relevance of individual features. Lexical similarity is used as a flexible approach to wildcards matching [10]. It controls the matching process semantically. It also supports incremental query transformations.

2.3.4. Spatial Similarity

Spatial similarity is determined based on the spatial relevance, i.e., placement in the same module, file or function, and is determined using a knowledge base [12]. *Part-of* relationships can be used to assess semantic similarity. For example, flow.ss (flow subsystem) is semantically similar to support.ss, because both subsystems are *part-of* the dead store module. Similarly, the memmap, memdsc, plshadow and refoccs data structures are semantically similar, because all are used in the dead store module (i.e, they are *part-of* ds.pl9).

3. Case Studies

This section describes how traditional retrieval techniques are extended using the proposed context-based similarity. Text-based search is enhanced by supporting lexical similarity, and by the ability to search in a neighborhood of a repository artifact. Faceted search is improved by supporting query transformations, and iterative browsing. Relevance of the retrieved artifact is determined with respect to the user task, preferences, and the current state of

[4] Although a phonetic similarity may be used, it would have a limited utility for software engineering.

the search. Next we discuss several tasks in which a software engineer may be involved. We show how context-based similarity supports the process. The studies presented are based on interviews of novice and expert users working on a specific project.

3.1. System Familiarization

Novice users do not have enough domain knowledge to support problem-oriented search through the repository. Thus, the method of interaction is scanning and the mode of retrieval is recognition. They need to acquire knowledge about basic data structures used in the project, and form an overall system design representation. As the users become more familiar with the studied system they acquire meta-information in the repository. The main objective for the retrieval system is to decrease the overall time needed for familiarization.

Since the novice user's goal is to understand a complex system they need to traverse the repository hierarchically. For these reasons, query-by-example, query-by-reformulation, and iterative browsing are suitable retrieval strategies for novices.

Finnigan et al. [12] describe scenarios, where a novice navigates through the repository by following links from a system architecture diagram, obtains a subsystem overview, and retrieves subsystem details by selecting one of its possible views. Similarity-based retrieval enhances the current tools by supporting approximate queries, and browsing through conceptually neighboring software artifacts.

The proposed context-based similarity retrieval system can further assist the user by suggesting query selection, by performing query transformations (relaxation or restriction), and by providing additional relevant information. This helps to both control the amount of information presented to the user, and explore the neighborhood of a particular software artifact (using semantic or spatial similarity as defined in Section 2.3). Both issues are crucial in helping a novice create a mental model of the system faster.

Assume the following query:

```
name: {dsinit},
formal arguments: {int, float},
return type: {float},
source location: {dsini}.
```

If the user wants to explore other relevant artifacts, then the query must be relaxed. The following relaxation strategies are available:

- *Lexical similarity* may be used to find lexically similar artifacts.
- *Reduction* can be used to explicitly remove attributes from the context, e.g., eliminate source location from the matching process. The other possibility is to dynamically reduce the number of required attributes, i.e., to apply *m_of_n matching*.
- G*eneralization* may be applied, by using *is-a,* and *part-of* hierarchies, or expansion of the set of allowed values.

For example, relaxing the constraint on the source location attribute can be generalized from {dsini} to {dead_store}, using the following piece of domain information: dsini *part-of* dead_store. Yet another possibility is to expand the set of allowed values, e.g., we can change return type : {float} to return type : {int, float}.

3.2. Program Development

Developing a software system requires familiarity with its code. Since the source code for a large system is usually stored in a hierarchical tree structure, navigation through it may be difficult. It is not uncommon to "spend one to three days looking for a definition, only to write one new line of code" [32]. It was estimated that in about "three quarters of a year, one month is spend on searching for information" [32].

Another problem with developing a large system is that many developers are involved. This means, that code changes are frequent. On a specific project it was observed that "in about 2.5 hours, some 50 files were modified" [32]. Thus, a desirable retrieval mechanism should allow for *imprecise query* specification and for similarity-based retrieval. This can help locate relevant artifacts without overloading a user with irrelevant information. Due to the frequency of code-change, running a cross-reference utility once a day might not provide sufficient information. However, similarity-based retrieval finds relevant matches using lexical similarity or generalization.

Generally, queries are direct, e.g., find a specific variable definition. However, there is also a need for imprecise queries – to deal with the problem of change and lack of domain knowledge, and similarity-based retrieval – to consider conceptually neighboring software artifacts.

Consider an example where a developer is looking for a function that is probably called ds_mrgs, since it is part of the dead store module. Using *lexical similarity* the user finds that there is a function called dsmrgs. Another possibility is to use *functional similarity* to locate ds_mrgs using its signature:

⟨name: D_{name},
formal arguments: (int, float, bit value),
return type: integer,
source location: $D_{location}$⟩.

Let us assume that the user found the specified function and thus name: dsmrgs and source location: dsini (the remaining parts of the signature remain unchanged). This forms an initial query-by-example. If the user needs to locate other similar functions, there are several options for query relaxation. First, reduction can be used: it is sufficient if either name or source location is matched, provided that both formal arguments and return type match. In addition, reduction can be applied to formal arguments – it would be sufficient if only *two_out_of_three* arguments match. If the user reduces the signature to source location: dsini, then in effect spatial similarity is used.

Second, generalization can be used: return type: integer can be changed to return type: number. Alternatively, formal arguments: (int, float, bit value) can be changed to formal arguments: (number, number, bit value). Finally, return type can be changed to return type: integer, Boolean, meaning that either integer or Boolean type is considered a match.

Domain knowledge may be used to improve the query processing, by improving query transformations. For example, the tree structure of the source code is built with certain preferences. Also, another development team may use .ch files for in-line definitions. There is an interesting performance issue: if the .ch file is included in the header file (class definitions) then it results in a fast executable but a slower compilation; if it is included in a source file, then the compilation is faster but it results in slower executable. Other types of preferences come naturally from the task under consideration. For example, when a developer needs to find a variable, (s)he usually looks for a store operation first, then for a fetch. Thus, the system may predict this and pre-load information about the fetch operation right after retrieving a store operation.

Similarity matching can also be used in situations when a query is specified imprecisely. This may include situations, where the type of a variable is known but not its name. The retrieval tool needs to find the definition of all variables with a particular type, but can possibly order them based on their location in the tree. This may affect how relevant the variable from a particular module is in a given situation. This can be implemented via signature matching [27], as exemplified in Section 3.3.

Since many functions, class and variable definitions are defined using inheritance, an obvious question is how many levels should be traversed in order to answer the query. Because different tasks require different depth, the number of levels to be traversed should be selectable.

3.3. Migration and Reverse Engineering

Migration of the software system includes its porting to another computer architecture or to another language. During this process, an exact copy of the system in another language may be produced (transliteration process). Alternatively, a higher level migration, such as design migration and architecture migration can be done.

Transliteration is the process where code written in one language is directly translated into another language. The process may be completely automated or it may be semi-automatic. This may result in inefficient code since different languages support various constructs with variable efficiency. A more difficult approach is to take data structures into consideration and to optimize them during system migration. More complicated issues arise when change of they system architecture system is required.

Migration and reverse engineering programmers face the problem of finding variable declarations efficiently and reliably. A recall-oriented retrieval process is needed, since the failure to retrieve all relevant artifacts may lead to incorrect decisions. Precision must also be high, to reduce information load on the user. Retrieval may be complicated by the fact that a function call might look identical to a variable (e.g., PL/I dialect), and the two may be distinguishable only from their declaration. Relevant queries include "Find a variable declaration", "Find a variable usage", and "Find a function with a parameter of a specific data type".

Migration software engineers are supported using a two-phase approach. (1) All relevant information from the repository is retrieved. (2) The query is iteratively transformed, so irrelevant artifacts are removed. Another strategy is to find a relevant artifact via query-by-example and consider other related artifacts in the repository. Consider a signature of the following function:

⟨name : dsinit,
formal arguments : (int, float),
return type : float,
source location : dsini⟩.

Reduction with generalization can be applied to find other functions with both their arguments and return types being equal. Query reduction results in the following:

formal arguments : {int, float},
return type : {float}.

Generalization may change the first argument of the formal arguments from int to number:

formal arguments : {number, float},
return type : {float}.

3.4. Functional Description Repository

This section describes preliminary performance evaluation of the proposed similarity-based retrieval system. Our claim is that similarity-based retrieval improves precision, and supports iterative browsing. Our previous studies suggest that the claim holds [19, 21]. Here we consider a functional description repository of basic data structures, similar to the one in [8, 10].

A functional description is a repository artifact described by a set of features and organized into categories. Each feature is a triplet: ⟨verb, noun, weight⟩, where verb is the operation, noun is the object and weight indicates importance of the feature in the given functional description. A sample functional description with two features can be defined as follows:

(⟨verb : insert, noun : node, weight : low⟩,
⟨verb : delete, noun : node, weight : high⟩).

During retrieval, if no relevant artifacts are returned for the specified functional description, the query might be transformed, as described in Section 2.1.2. First, reduction is applied, and then followed by generalization. Generalization uses *part-of* and *is-a* hierarchies of terms, e.g., node *part-of* btree, element *part-of* list, btree *is-a* tree, etc. Such hierarchies make it possible to both specify imprecise queries and to explore a neighborhood of the currently retrieved artifacts. For example, after retrieving a functional description, such as ⟨verb : insert, noun : node, weight : low⟩, one could explore its neighborhood, i.e., define a context that would retrieve other similar artifacts from the repository with respect to certain features. First, consider an operation – find other functional descriptions with the same operation:

verb : {insert},
noun : {element},
weight : {high}.

Second, find functional descriptions with different operation on the object that is in addition similar to a given description:

verb : {delete},
noun : {node},
weight : {medium}.

During evaluation, a functional description was selected at random and submitted as in query-by-example (leave-one-method). We have evaluated recall and precision after averaging results over twenty random tests. Using explicitly stated context and query transformations we could control the relevance (quality) and quantity of retrieved artifacts. By selecting reduction, generalization or both, we simulated recall- and precision-oriented retrieval, and recall-oriented retrieval with iterative precision improvement. Using the last approach yielded 100% recall rate with perfect precision. These results are consistent with our previous evaluations [19, 21], but since the repository of functional descriptions is rather small (only 150 data structures) and simplistic, we intend to run full-scale studies on a larger and more complex software repository.

Performance improvement over results presented in [8, 10] was achieved for two reasons. First, context-based similarity assessment keeps semantic information about distance among artifacts, which enables better control over relevant and irrelevant artifacts. Second, a combination of the retrieval system with a high recall, and iterative browsing that removes irrelevant artifacts increases precision. The main distinguishing factor, when comparing the proposed context-based similarity to systems such as [34] and [10], is semantic information that is part of the similarity assessment process.

Besides task performance, we have also tested scalability of the system, albeit on different domains so far. We have analytically and experimentally measured dependency of query response time on the repository size, query complexity and context transformation strategy. The results we have obtained show that the incremental approach and context-based similarity are superior to traditional retrieval, both in terms of quality of the retrieved artifacts [19] and speed in which the answer is obtained [20].

4. Conclusions

The work presented here is relevant to other aspects of a software repository: repository architecture [12], the process of building complex decision-support systems [23], and user modeling [9]. As a next step, we plan to integrate individual components of a software repository and to conduct a more comprehensive performance evaluation which will include task performance, system scalability and user acceptance. User studies will indicate whether our goal of supporting diverse user models has been achieved.

In addition, we plan to include additional similarity measures. First, *purposeful similarity* could provide for similarity due to the purpose of the function or a variable (e.g., used for I/O, graphics rendering). Second, *dependency* could locate a data-flow dependency, and relevancy based on version or time interval. Context-based similarity could be also used to support dynamic and interactive program slicing. Third, *correspondence* would cover relationships of individual software artifacts during migration process.

Context-based similarity can be used to solve other problems that are beyond the scope of this paper. A natural application is repository clustering [5, 8, 34]. The proposed context-based similarity assessment can also be used to organize knowledge by clustering the repository content into equivalence classes, i.e., into classes of artifacts that share similarity with respect to the given context [21]. Similar to retrieval, context can be inferred from either the active user model, or the task being solved. The user can manually alter the context at all times.

Context-based similarity enables fuzzy queries and similarity-based retrieval [19]. For related work see [11, 13, 40, 39]. Case-based reasoning can be applied in user modeling [9]. It extends traditional approaches by representing users as cases. User profiles, tasks and actions guide similarity-based retrieval. We can also use case-based reasoning to evaluate performance, since after we accumulate cases, we can estimate the execution time of the query based on its complexity and also on the network load [9].

A contribution of this paper is in extending retrieval capabilities of the Bookshelf repository, by using context-based similarity assessment algorithm. Flexibility and efficiency of iterative browsing and retrieval enables effective use of such a repository to support novice and expert users during familiarization, program development, migration and reverse engineering tasks.

Although preliminary results are promising, further work is needed to make the system usable to actual users. We are working on better integration of the retrieval module with the Bookshelf repository and with the working environment of potential users. In addition, we plan to compare performance of users alone and users with the proposed system on the retrieval scenarios presented above.

Acknowledgment

The Centre for Advanced Studies of IBM Canada, the Canadian Consortium for Software Engineering Research, and the Information Technology Research Centre of Ontario generously supported this research.

The author is indebted to J. Mylopoulos, J. Glasgow and N. Zahariadis for helpful comments and suggestions and to the TOBEY group within the IBM Toronto lab for sharing their technical knowledge and experience. Suggestions and comments of anonymous reviewers are also appreciated.

References

[1] N. J. Belkin, C. Cool, A. Stein and U. Thiel. Cases, scripts, and information-seeking

strategies: On the design of interactive information retrieval systems. *Expert Systems with Applications,* 9(3): 379-395, 1995.

[2] G. Canfora, A. Cimitile and M. Munro. An improved algorithm for identifying objects in code. *Software-Practice and Experience.* 26(1): 25-48, 1996.

[3] P. Cunningham, A. Bonzano and B. Smyth. An Incremental Case Retrieval Mechanism for Diagnosis, *Technical Report TCD-CS-95-01*, Trinity College, Dublin, 1995.

[4] P. Constantopoulos, M. Jarke, J. Mylopoulos, and Y. Vassiliou. The software information base: A server for reuse. *VLDB Journal,* 4, 1995.

[5] C. Carpineto and G. Romano. A lattice conceptual clustering system and its application to browsing retrieval. *Machine Learning.* 24(2): 95-122, 1996.

[6] W. W. Chu, H. Yang, K. Chiang, M. Minock, G. Chow, C. Larson. CoBase: A scalable and extensible cooperative information system. *Journal of Intelligent Information Systems*, (6), 1996.

[7] R. P. D'iaz and P. Freeman. Classifying software for reusability. *IEEE Software*, 4(16):6-16, 1987.

[8] K. S. Daudjee and A. A. Toptsis. Automatic organization of reusable software components in a multidimensional space. In *WITS-94*, pages 11-20, Vancouver, BC, 1994.

[9] B. Errico and I. Jurisica. CaBUMA: A case-based approach to user modeling. In preparation.

[10] M. Fäustle, G. Fugini and E. Damiani. Retrieval of reusable components using functional similarity. *Software-Practice and Exprience.* 26(5): 491-530, 1996.

[11] C. Fernandezchamizo, P. A. Gonzalezcalero, L. Hernandezyanez and A. Urechbaque. Case-based retrieval of software components. *Expert Systems with Applications*, 9(3): 397-405, 1995.

[12] P. Finnigan, R. Holt, I. Kalas, H. Müller, J. Mylopoulos, S. Perelgut, M. Stanley, K. Wong, S. Kerr. The Software Bookshelf. *IBM Systems Journal*. Accepted for publication.

[13] G. Fouque and S. Matwin. Compositional software reuse with case-based reasoning. In *CAIA-93*, 1993.

[14] T. Gaasterland. Restricting query relaxation through user constraints. In Proc. *International Conf. on Intelligent and Cooperative Information Systems,* pages 359-366, Rotterdam, The Netherlands, 1993

[15] T. Gaasterland, P. Godfrey and J. Minker. Relaxation as a platform of cooperative answering. In. *Proc. International Workshop on Nonstandard Answers and Queries.* Toulouse, France, 1991.

[16] M. R. Girardi and B. Ibrahim. A similarity measure for retrieving software artifacts. Technical Report, Universtiy of Geneva, Centre Universitaire d`Informatique, 1994.

[17] P. Garg and W. Scacchi. ISHYS: Designing an intelligent software hypertext system, *IEEE Expert*, pages 52-82, 1989.

[18] S. Henninger. A case-based approach to developing knowledge for software development. In The *3rd Workshop on AI and Software Engineering: Breaking the Mold. IJCAI-95*, Montreal, Quebec, 1995.

[19] I. Jurisica and J. Glasgow. Improving performance of case-based classification using context-based relevancy. *International Journal of Artificial Intelligence Tools. Special Issue of IEEE ITCAI-96 Best Papers*. 6(3&4), In Press. 1997.

[20] I. Jurisica and J. Glasgow. An efficient approach to iterative browsing and retrieval for case-based reasoning. Submitted. 1997.

[21] I. Jurisica, J. Mylopoulos, J. Glasgow, H. Shapiro, and R. F. Casper. Case-based reasoning in IVF: Prediction and knowledge mining. *AI in Medicine*, 1997. In Press.

[22] H. V. Jagadish, A. O. Mendelzon, and T. Milo. Similarity-based queries. *PODS*, 1995.

[23] I. Jurisica and B. Nixon. Building quality into case-based reasoning systems. Submitted, 1997.

[24] I. Jurisica. How to retrieve relevant information? In *[Gre94]*, pages 101-104, 1994.

[25] I. Jurisica. A similarity-based retrieval tool for software repositories. In The *3rd Workshop on AI and Software Engineering:*

Breaking the Mold. IJCAI-95, Montreal, Quebec, 1995.

[26] I. Jurisica. Supporting flexibility. A case-based reasoning approach. In *The AAAI Fall Symposium. Flexible Computation in Intelligent Systems: Results, Issues, and Opportunities*, Cambridge, MA, 1996.

[27] I. Kalas. *Personal communication on bookshelf architecture and bookshelf query language.* IBM Toronto Lab, Centre for Advanced Studies, Toronto, Ontario, 1997.

[28] T. Kokeny. Constraint satisfaction problems with order-sorted domains. *International Journal on Artificial Intelligence Tools,* 4(1&2): 55-72, 1995.

[29] D. Lauzon and T. Rose. Task-oriented and similarity-based retrieval. In *Conf. on Knowledge-Based Software Engineering*, 1994.

[30] H.Y. Lee and M.T. Harandi. An analogy-based retrieval mechanism for software design reuse. In Proc. *8th Knowledge-Based Software Engineering Conf.*, pages 152-159, Chicago, IL, 1993.

[31] X. Li, N.S. Hall and G. W. Humphreys. Discrete distance and similarity measure for pattern candidate selection. *Pattern Recognition,* 26(6): 843-851, 1993.

[32] I. McIntosh. *Personal communication. Useful tasks for Bookshelf.* IBM Toronto Lab, CAS, Toronto, Ontario, 1997.

[33] J. Mylopoulos, A. Borgida, M. Jarke, and M. Koubarakis. *Telos*: Representing knowledge about information systems. *ACM Trans. Information System,* 8(4): 325-362, 1990.

[34] Y. Maarek, D. Berry, and G. Kaiser. An information retrieval approach to automatically constructing software libraries. *IEEE Tr. on Software Engineering*, 17(8):800-813, 1991.

[35] J. Ortega. On the informativeness of the DNA promoter sequences domains theory. *Journal of Artificial Intelligence Research,* 2: 361-367, 1995.

[36] E. Ostertag, J. Hendler, R. P. D'iaz, and C. Braun. Computing similarity in a reuse library system: An AI-based approach. *ACM Trans. on Software Engineering and Methodology*, 3(1):205-228, 1992.

[37] M. Rittri. Using types as search keys in functional libraries. *Journal of Functional Programming*, 1(1), 1991.

[38] E. H. Ruspini. Approximate reasoning: Past, present, future. *Information Sciences*, 57-58:297-317, 1991.

[39] R. W. Schwanke. An intelligent tool for re-engineering software modularity. In Proc. *14th Conference on Software Engineering*, pages 83-92, Austin, TX, 1991.

[40] George Spanoudakis and Panos Constantopoulos. Similarity for analogical software reuse: A computational model. *ECAI*, pages 18-22, 1994.

[41] P. R. Thagard, K. J. Holyoak, G. Nelson and D. Gotchfeld. Analog retrieval by constraint satisfaction. *Artificial Intelligence,* 46: 259-310, 1990.

[42] M. Tedjini, I. Thomas, G. Benoliel, F. Gallo, and R. Minot. A query service for a software engineering database system. In Proc. *4th Symposium on Software Development Environments*, pages 238-248, 1990.

[43] P. Termsinsuwan, Z. X. Cheng and N. Shiratori. A new approach to ADT specification support based on reuse of similar ADT by the application of case-based reasoning. *Information and Software Technology.* 38(9):555-568, 1996.

[44] A. Tversky. Features of similarity. *Psychological Review*, 84(4): 327-352, 1977.

[45] M. A. Vila, J. C. Cubero, J. M. Medina and O. Pons. A conceptual approach for dealing with imprecision and uncertainty in object-based data models. *International Journal of Intelligent Systems.* 11, pages 791-806, 1996.