A survey of software reuse libraries

A. Mili [a,*], R. Mili [b] and R.T. Mittermeir [c]

[a] *The Institute for Software Research, 1000 Technology Drive, Fairmont, WV 25554, USA*
E-mail: amili@cs.wvu.edu

[b] *School of Engineering and Computer Science, University of Texas at Dallas, Richardson, TX 75083, USA*
E-mail: rmili@utdallas.edu

[c] *Institüt für Informatik, Universität Klagenfurt, A-9022 Klagenfurt, Austria*
E-mail: roland@ifi.uni-klu.ac.at

The study of storage and retrieval methods of software assets in software libraries gives rise to a number of paradoxes: While this subject has been under investigation for nearly two decades, it still remains an active area of research in software reuse and software engineering; this can be explained by the observation that new technologies (such as the internet, the world wide web, object-oriented programming) keep opening new opportunities for better asset packaging, better library organizations, and larger scale libraries – thereby posing new technical challenges. Also, while many sophisticated solutions have been proposed to this problem, the state of the practice in software reuse is characterized by the use of ad-hoc, low-tech methods; this can be explained by the observation that most existing solutions are either too ineffective to be useful or too intractable to be usable. Finally, while it is difficult to imagine a successful software reuse program without a sophisticated, well-tuned, systematic procedure for software component storage and retrieval, it seems many successful software reuse experiments rely on trivial methods of component storage and retrieval; this can be explained by the observation that, in the current state of the practice, software libraries are not the bottleneck of the software reuse process. This paper presents a survey of methods of storage and retrieval of software assets in software libraries. In addition to a review of existing research efforts, the paper makes two contributions. First, a definition of (presumably) orthogonal attributes of storage and retrieval methods; these attributes are used, in turn, to classify existing methods into six broad classes. Second, a definition of (presumably) orthogonal assessment criteria, which include technical, managerial and human factors; these criteria afford us an exhaustive and uniform basis for assessing and comparing individual methods and classes of methods.

1. Introduction

The storage and retrieval of reusable assets in a software library has been the subject of active research in the past. While a large number of solutions have been proposed, it is difficult to consider that this issue has been resolved satisfactorily. On the contrary, the widespread use of the internet and the advent of new programming languages and paradigms (such as Java) provides new opportunities and challenges for

* This paper was written while the first author was at The University of Ottawa, Ottawa, Canada.

software reuse [Fickas 1997; Browne and Moore 1997; Faison 1997] calling for further research. Addressing these challenges, one has to be aware of the basic representation issues that architects of a library of reusable components are confronted with. As with general (book) libraries, there is a hierarchy of abstractions a person seeking a component (or a book) has to perform. The librarian has to anticipate these abstractions in the best possible way. That this is a non-trivial task can be inferred from the existence of multiple levels of abstraction:

- *Utility level.* The user of the library is seeking some particular item of information that might be contained in a book or some service that might be rendered by a software component.
- *Artifact level.* To obtain this utility, one has to get an appropriately written (style, level of detail, language, etc.) book or an integrable or otherwise usable software component.
- *Representation level.* In order to find the artifact from which the expected utility is derived, one performs a search over some representations of this artifact. Some of these representations (e.g., the book number) have just a system internal structural purpose. Others, like the authors name or a subject index are mechanisms to establish a link between the utility level and the artifact level. This link is needed with books as much as with software, since the information contained in the artifact itself would be in general too voluminous and too detailed to be searched directly.

The three levels mentioned above highlight the similarity between conventional libraries and libraries of reusable software assets, whether the latter take the form of code or the form of non-executable artifacts resulting from some, and useful for some other, software development process. In both cases, (re-)users have to derive abstractions of what they actually want in order to find the artifacts that are potentially useful. As can be seen from the different information content of an authors-index and a subject-index, different abstractions (and their representations) perform quite differently as links. But another problem has to be borne in mind: unless one can do plain text search, with the expectation that the artifact itself furnishes its representation on demand, the indexing has to be done by a librarian according to fixed rules so that patrons of the library can be confident of consistent representations of books and concepts in the index. Likewise, users of a software library have to be confident of consistent classifications and representations. This focus on consistency reduces the problems resulting from the fact that the librarian's representation is made already when the asset is incorporated into the library. It is made in full knowledge of the artifact at hand. The user's representation is made much later, however – when the artifact is checked out. But the user's representation is made in ignorance of the artifact. It is made in full knowledge of the desired/expected utility though. This separation in time and in the basis of abstractions might be the key reason that makes software storage and retrieval a challenging issue; an issue that is much more similar to issues dealt with in library science and information retrieval than in conventional

database research. A more detailed discussion of these links between library science and software reuse can be found in [Frakes and Gandel 1990].

In order to synthesize past achievements and define a framework for future work, we present in this paper a critical survey, as well as a classification, of existing methods. Section 2 presents a terminology of software storage and retrieval, that we use throughout the paper; then it presents a set of criteria that can be used to assess methods of storage and retrieval of software assets. In addition, we propose a set of attributes that can be used to characterize storage and retrieval methods, and use these attributes as a basis to classify methods according to their characteristics. Specifically, we have divided existing methods into six broad families, which are discussed in sections 3–8. Each of these sections is organized into three subsections: *Literature survey*, where we briefly review the main methods that fall in this category; *Characterization*, where we attempt to give a general characterization of the proposed methods in terms of the attributes discussed in section 2; and *Assessment*, where we assess the methods of the given family in terms of the criteria set forth in section 2. In subsections *Literature survey*, we attempt to highlight how each method of the family differs from the others; in subsections *Characterization*, we attempt to highlight what the methods of the same family have in common (and, a posteriori, why we placed them in the same family); and in subsections *Assessment*, we highlight how the methods of each family fare by comparison with methods of other families. In the conclusion (section 9) we summarize our contributions, present a synthetic assessment of all the families of methods, then briefly review some of the earlier surveys of software libraries.

Traditionally, whenever a new method of software components storage and retrieval is introduced, it is compared against a selection of existing methods, most of them are so different that the comparison is hardly instructive. In addition, in the absence of an exhaustive, uniform, set of criteria, the methods are compared in an informal, incomplete manner. Our aim is to streamline this process in two ways. First, comparison to existing methods would consist of identifying the class to which the proposed method belongs, then comparing the method to other members of the class. Second, assessment of the proposed method can be focused by discussing how the method fares with respect to each of the evaluation criteria, and how it compares with alternatives.

2. Classifying and assessing software libraries

In this section we describe the criteria and attributes used for assessing and characterizing methods for the storage and retrieval of reusable software components. After presenting some terminology, we discuss assessment criteria and define characterizing attributes for storage and retrieval methods.

2.1. Terminology for storage and retrieval

A *software library* is a set of software assets that are maintained by an organization for possible *browsing* and/or *retrieval*; while *software assets* are typically source code units, they could also be other assets, such as specifications, designs, documentation, or test data. Also, while the most common application of software libraries is software reuse, it is not necessarily the only application [Zaremski and Wing 1995a]. The difference between *browsing* and *retrieval* is that while the latter consists in identifying and extracting assets that satisfy a predefined *matching criterion*, the former consists in inspecting assets for possible extraction, without predefined criterion. In this paper, we focus our attention on *retrieval*, although we may occasionally discuss browsing techniques.

Because retrieval depends on matching a candidate asset against a user *query*, the representation of both the query and the asset is an important consideration. Queries can be represented in a number of different ways, including:

- natural language patterns and templates (for *information retrieval methods*, section 3);
- lists of keywords (for *descriptive methods*, section 4);
- sample input/output data (for *operational semantics methods*, section 5);
- input/output signature or functional description (for *denotational semantics methods*, section 6);
- designs or program patterns (for *structural methods*, section 8).

Assets have an even wider range of representation than queries, since they usually include a reference to the actual asset. The representation of the asset is typically substantially more abstract than the asset itself, and is made purposefully abstract to capture the important features of the asset while overlooking minor or irrelevant details; in the information retrieval literature [Frakes and Baeza-Yates 1992; Salton 1975], the representation of the asset is referred to as the asset's *surrogate*. For the sake of efficiency, the matching criterion between a query and an asset typically involves the asset's surrogate, rather than the asset itself.

We distinguish between two aspects in the activity of *asset retrieval*, whereby the library is scanned for identification of *relevant* assets: *navigation*, which determines which assets are visited, and eventually in what order they are visited; and *matching*, which determines the condition under which a visited asset is selected. Brute force navigation consists in visiting all the assets of the library in an arbitrary order; this occurs when the *storage structure* of the library is a flat organization. More elaborate navigation methods are possible when the storage structure has a meaningful organization. Matching consists in testing a condition that involves a given asset (or its surrogate) to determine whether the asset must be selected. We distinguish between two conditions: the *relevance condition* (also called the relevance *criterion*), which is the condition under which an asset is considered to be relevant with respect to the

pragmatics of a query; and the *matching condition* (also called the matching *criterion*), which is the condition under which an asset is actually selected. In practice, these two conditions may be different: if the relevance condition is too complex to be automatically tested or if the asset's surrogate is too abstract to afford a precise characterization of relevant assets, one may choose a matching condition that is weaker than the relevance condition; on the other hand, whenever a high degree of retrieval precision is required, one may select a matching criterion that is stronger than (i.e., logically implies) the relevance criterion. Under these conditions, the matching condition can be seen as an implementation of the relevance criterion.

We distinguish between (at least) two goals of an *asset retrieval* operation: *exact retrieval*, whose purpose is to identify library assets that are correct with respect to a submitted query; *approximate retrieval*, whose purpose is to identify library assets that can be modified with minimal effort to satisfy a submitted query. Exact retrieval fits in the life-cycle of *black box reuse*, whose phases are: *exact retrieval*, whereby correct assets are identified and retrieved; *assessment*, whereby retrieved assets are evaluated with respect to the query to select that which provides the best fit; *instantiation*, whereby the selected asset is duly specialized to fit the purpose of the query at hand; *integration*, whereby the instantiated asset is integrated into its host system and submitted to some form of integration testing. Approximate retrieval fits in the life-cycle of *white box reuse*, whose phases are: *approximate retrieval*, whereby assets that approximate the query are identified and retrieved; *assessment*, whereby retrieved assets are evaluated with respect to the query to select that which minimizes the expected modification effort; *modification*, whereby the selected asset is duly modified to fit the purpose of the query at hand; *integration*, whereby the instantiated asset is integrated into its host system and duly tested. We distinguish between two patterns of program modification: *generative modification*, which consists in reusing the design of an asset while rewriting the lower levels of its hierarchical structure; *compositional modification*, which consists in using the building blocks of the retrieved asset and combining them in a new design structure to satisfy the query at hand.

It may be worthwhile to illustrate the distinction between *retrieval goal*, *relevance criterion* and *matching condition* by means of an example. If we consider the method of software storage and retrieval by signature matching, we could conceivably define two possible retrieval goals: either our retrieval goal is that retrieved components be correct with respect to the submitted query, or it is that retrieved components have the same signature as the submitted query. If we are interested in a component that computes the depth (height) of a binary tree we may submit the signature

binarytree, naturalnumbers.

For the sake of argument, assume that the method returns a component that computes the depth of a binary tree and a component that computes the number of nodes of a binary tree. Under the first retrieval goal, only one component is deemed relevant (the first), whereas under the second, both are deemed relevant. Hence precision and recall (which count the number of relevant assets that are or are not retrieved) cannot

be defined unless a retrieval goal is agreed upon. On the other hand, to illustrate the distinction between *relevance criterion* and *matching condition*, consider the storage and retrieval method that operates by behavioral sampling: the relevance criterion is typically that the candidate component be correct with respect to the reuser's intent, and the matching condition is (only) that the candidate component behave satisfactorily on the data sample provided by the reuser; clearly, these are different conditions.

2.2. Assessment criteria

Given that we discuss a wide range of storage and retrieval methods in this paper, it is important that we define a uniform set of criteria that we use to assess and compare these methods; the purpose of this section is to introduce the criteria that we use to this effect. We will discuss in turn technical criteria, managerial criteria, then human criteria; for each criterion, we will attempt to present a quantitative metric to measure it. Whenever it appears that we cannot estimate the quantitative metric in practice, we derive a discrete 5-rating scale instead, which is usually adequate for the purposes of our discussions. We do nevertheless maintain the quantitative metric, because first we do not want our inability to estimate a quantity to affect our definition of the quantity, and second we feel that the precise definition serves to focus our ideas even when we cannot use it in practice.

2.2.1. Technical criteria

We have identified six technical evaluation criteria, which are discussed in turn below.

1. Precision. The precision of a retrieval algorithm is the ratio of relevant retrieved assets over the total number of retrieved assets; this is a number that ranges between 0 and 1. Under the hypothesis that all library assets are visited (e.g., exhaustive navigation), we get perfect precision ($= 1$) whenever the matching criterion logically implies the relevance criterion. This can be achieved in particular by letting the matching criterion be **false**, which means that no assets are returned (hence no irrelevant assets are returned). Because all we are doing in this survey is hypothesizing about the performance of retrieval algorithms, it is impossible to assign actual numeric values for precision; we will, instead, assign values in a discrete five-rating scale (*Very Low* (VL), *Low* (L), *Medium* (M), *High* (H), *Very High* (VH)), where VH refers to perfect precision and VL refers to very poor precision.

2. Recall. The recall of a retrieval algorithm is the ratio of relevant retrieved assets over the total number of relevant assets in the library; this is a number that ranges between 0 and 1. Under the hypothesis that all library assets are visited (e.g., exhaustive navigation), we get perfect recall ($= 1$) whenever the relevance criterion logically implies the matching criterion. This can be achieved in particular by letting the matching criterion be **true**, which means that all library assets are returned (hence no relevant assets are forgotten). For the same reasons as with precision, we cannot, in general,

assign actual numeric values to candidate algorithms; we will, instead, use a five-rating scale where VH stands for perfect recall and VL stands for very poor recall.

3. Coverage ratio. The coverage ratio of a retrieval algorithm is the average number of assets that are visited over the total size of the library; this is a number that ranges between 0 and 1. The brute force exhaustive navigation produces a coverage ratio of 1, but more elaborate algorithms may take advantage of the storage structure to exclude portions of the library from consideration. Ideally, in order to preserve recall, they must ensure that no excluded portion of the library contains any relevant assets. Again, we use a five-rating scale rather than a numeric scale, where VH stands for an exhaustive coverage and VL stands for a very selective coverage.

4. Time complexity per match. The time complexity of a match measures the number of computation steps that are required to match the query against a library asset; we represent this measure by the O(N) notation, where N is some measure of size of the query. To simplify comparisons, we define the following scale for this measure:

Very Low	Low	Medium	High	Very High
Constant	Linear	Polynomial	Exponential	Unbounded

This measure can be used together with coverage ratio to estimate the time performance of a retrieval algorithm. The mean time it takes to perform a retrieval in a library of size L that has a coverage ratio of *CR* and a time complexity of *TC* is

$$L \times CR \times TC,$$

or, if we wish to derive a size-independent measure, $CR \times TC$.

5. Logical complexity per match. The performance of a retrieval method has to be considered against the complexity of performing a match under this method: the more complex the match, the more one would expect from the method (in terms, e.g., of precision and recall). Also, the logical complexity of a retrieval method is important because it is a determining factor of the method's automation potential. We have derived a five-rating scale for logical complexity:

Very Low	Low	Medium	High	Very High
Simple Boolean	Compound Boolean	Simple predicate	Compound predicate	Second order predicate

6. Automation/automation potential. An important criterion in the success of a storage and retrieval method is the potential that the method be automated. We have derived a five-rating scale to this effect:

Very Low	Low	Medium	High	Very High
Non-automatable	Requires a major effort	Requires non-trivial effort	Automatable with some effort	Trivially automatable

This scale is highly subjective but is, nevertheless, adequate for our purposes.

2.2.2. Managerial criteria

We have identified the following managerial criteria, which we will use subsequently to assess storage/retrieval methods.

1. Investment cost. This criterion reflects the cost of setting up a software library that implements the proposed method, pro-rated to the size of the library. Because we expect that the largest portion of this cost stems from manpower effort, we propose to measure the cost in person-months. For the sake of this survey, we use a five-value rating scale where VL refers to a very low investment cost (can be amortized in the short run, even with limited use) and VH refers to a very high investment cost (can be amortized only under the condition of intensive long term use).

2. Operating cost. This criterion reflects the yearly cost of operating a software library, prorated to the size of the library. For the reasons that we discussed above, we represent cost in person-months; because time appears in both the numerator and the denominator of our ratio, it can actually be canceled, and we are left with a ratio of Full Time Software Person (FSP) [Boehm 1981] per 1000 assets:

$$\frac{\text{FSP}}{\text{1000 Assets}}.$$

For the sake of this survey, we use a five-value rating scale where VL refers to a very low operating cost (a fraction of a Full Time Software Person for a large library) and VH refers to a very high operating cost (more than one Full Time Software Person for a few hundred components).

3. Pervasiveness. This criterion reflects to what extent the proposed method is widely used in research and development. A method gets a high rating either because several research laboratories are investigating it or because several development shops are using it in practice. The five-rating scale that we propose for this criterion is the following:

Very Low	Low	Medium	High	Very High
Not used	Barely used	Some use	Widely used	Routine use

4. State of development. This criterion reflects the state of development of the method at hand: whether it is a mere speculative idea or a fully operational industrial product. The five-rating scale that we propose for this criterion is the following:

Very Low	Low	Medium	High	Very High
Speculative idea	Laboratory prototype	Experimental product	Industrial product	Fully supported industrial product

2.2.3. Human criteria

We have identified two human criteria that we can use to evaluate storage and retrieval methods for software libraries.

- *Difficulty of use.* This criterion accounts for the fact that the various methods provide varying intellectual challenges to their users. We have derived the following five-rating scale to assess difficulty of use:

Very Low	Low	Medium	High	Very High
Trivial	Easy	Nontrivial	Difficult	Very difficult

- *Transparency.* This criterion reflects to what extent the operation of the asset library depends on the user's understanding of how the retrieval algorithm works. We have derived the following five-rating scale to assess transparency:

Very Low	Low	Medium	High	Very High
Detailed knowledge	Good knowledge	Some cognizance	Slight cognizance	Perfect transparency

Occasionally, we may find that we are unable to meaningfully assign a rating to a particular class of methods for a particular criterion; under such circumstances, we will assign the value U, to stand for *Unknown.*

2.3. Characterizing storage and retrieval methods

Methods for the storage and retrieval of software assets in software libraries abound in the literature. In order to understand these methods and have a sound basis for classifying them, we characterize each method by a number of (nearly) orthogonal attributes. These attributes are presented and discussed in this section.

1. Nature of the asset. The most important feature of a software library is, of course, the nature of assets that are stored therein. The most typical asset is code (be it source code or executable code) but other kinds of assets are also possible: specifications, designs, test data, documentation, etc. Some library methods are restrictive, in the sense that they work for a single kind of asset, whereas others may work for a wide range of assets.

2. Scope of the library. Another crucial feature of a software library is the scope of the library: whether the library is expected to be used within a single project, within an organization, or on a larger scale. In order to use a software library effectively, a reuser must share some common knowledge with the maintainer of the library (pertaining, e.g., to the interpretation of terminology, the representation of assets, the form and meaning of error messages, etc.), and with other users. If this common knowledge is general (so that any reuser is likely to have it), then one expects the library to have a large scope; the more specialized this knowledge, the smaller the scope.

3. Query representation. A software library can be characterized by the form that queries submitted to the library must take. Among possible options we may mention: a formal functional specification, a signature specification, a behavioral sample, a natural language query, or a set of keywords.

4. Asset representation. The representation of assets is an important feature of a library, not only because it dictates what form user queries take, but also because it determines how retrieval is performed. In a perfectly transparent library, the representation of an asset is irrelevant to the user, but no library is perfectly transparent. Among the possible values of this attribute, we mention: formal specifications, signature specifications, set of keywords, the source text, the executable code, and requirements documentation.

5. Storage structure. The most common logical storage structure in software libraries is ... no structure at all: software assets are stored side by side with no ordering between them. While in traditional database systems entries are ordered by their identifying keys, it is difficult to define a general key that can be used to order software assets in a meaningful way. There are some exceptions: some libraries that are based on formal specifications order assets by the refinement ordering between their specifications; and AI-based software libraries define semantic links between assets.

6. Navigation scheme. This attribute is correlated to *storage structure*, because the storage structure determines to a large extent the navigation scheme of the method. In flat storage structures, the only possible pattern of navigation is brute force exhaustive search of all the entries. Whenever the assets are arranged on some non-trivial structure, this structure can be used to help orient the search towards those assets that are most likely to satisfy the query – and steer it clear from those that are known to be irrelevant or are thought unlikely to be relevant.

7. Retrieval goal. To fix our ideas, we discuss this feature in the context when the library assets are programs. In principle, the goal of a retrieval operation is to find one or several programs that are correct with respect to a given query. If this retrieval operation fails to turn up candidate programs, one may want to perform another retrieval operation, with the lesser ambition of finding programs that approximate the query, with the expectation that we must modify the (selected) retrieved programs. Depending on whether we are interested in generative modification or compositional modification, the goal of the retrieval operation changes considerably: under generative modification, we are interested in programs whose design can be adapted to solve the query; under compositional modification, we are interested in programs whose components can be combined to solve the query.

8. Relevance criterion. The relevance criterion defines under what condition a library asset is considered to be relevant for the submitted query with respect to the predefined retrieval goal. In reference to the discussion above about generative and compositional

Attributes	Characterization
Nature of asset	Source code, executable code, requirements specification, design description, test data, documentation, proof.
Scope of library	Within a project, across a program, across a product line, across multiple product lines, organization-wide world-wide.
Query representation	Functional specification, signature specification, keyword list, design pattern, behavioral sample.
Asset representation	Functional specification, signature specification, source code, executable code, requirements documentation, keywords.
Storage structure	Flat structure, hypertext links, refinement ordering, ordering by genericity.
Navigation scheme	Exhaustive linear scan, navigating hypertext links, navigating refinement relations.
Retrieval goal	Correctness, functional proximity, structural proximity.
Relevance criterion	Correctness, signature matching, minimizing functional distance, minimizing structural distance.
Matching criterion	Correctness formula, Signature identity, Signature refinement, Equality/Subsumption of keywords, Natural language analysis, Pattern recognition.

Figure 1. Attributes of a software library.

modification, observe that under generative modification, components are deemed relevant (or not) on the basis of their structure, whereas under compositional modification components are deemed relevant (or not) on the basis of their function.

9. Matching criterion. The matching criterion is the condition that we choose to check between the submitted query and a candidate library asset to decide whether the asset is relevant. Ideally the matching criterion should be equivalent to the relevance criterion, but it is not always so: if the asset's surrogate (the representation of the asset) is too abstract, and/or if the relevance criterion is too intractable, these two criteria may differ significantly. For example, if the relevance criterion is correctness of library assets with respect to the submitted query, and if library assets are represented by (arbitrarily abstract) functional specifications then it is tempting to let the matching criterion be the fact that the asset's surrogate (its specification) be a refinement of the query. This matching criterion is sufficient to ensure relevance (since it logically implies the relevance criterion) but is not necessary: it is possible that the asset does indeed satisfy all the requirements of some query, yet, because it is too abstract, the asset's surrogate (which does not record all the features of the asset) fails to refine the query; this is an instance where an asset satisfies the relevance criterion (since it is correct with respect to the query) but fails to satisfy the matching criterion (since its surrogate does not refine the query).

To summarize this section, we present in figure 1 a table of all the attributes discussed above, along with a tentative (not necessarily exhaustive) indication of the values that each attribute may take.

2.4. Classifying software libraries

Using the attributes listed in table 1, we have divided the existing library organizations into six classes, which we will discuss in the sequel. Each class corresponds to a distinct pattern of attribute values and includes all the library organizations that fit the pattern; whenever we discuss a class, we present the pattern of attributes that characterize it. We have identified six classes of storage/retrieval methods, which we present below, in the order of increasing technological sophistication.

1. Information retrieval methods. These are methods that depend on a textual analysis of software assets. It is important to acknowledge that the storage and retrieval of software assets is nothing but a specialized instance of information storage and retrieval. Hence it is important to discuss these methods, and possibly highlight their shortcomings: if traditional information retrieval methods were adequate in dealing with software assets, there would be little incentive to investigate other methods. This class of methods is the subject of section 3.

2. Descriptive methods. These are methods that depend on a textual description of software assets. While information retrieval methods represent assets by some form of text, descriptive methods rely on an abstract surrogate of the asset, typically a set of keywords, or a set of facet definitions. Also, while information retrieval methods select assets by attempting to *understand them* (in the sense of natural language processing), descriptive methods merely attempt to *characterize* candidate assets. This has a profound impact on the design as well as the performance of retrieval algorithms. This class of methods is the subject of section 4.

3. Operational semantics methods. These are methods that depend on the operational semantics of software assets. They can be applied to executable code, and proceed by matching candidate assets against a user query on the basis of the candidates' behavior on sample inputs. These constitute an elaboration on information retrieval methods, in the sense that they exploit a unique feature of software assets, namely their executability. This class of methods is the subject of section 5.

4. Denotational semantics methods. These are methods that depend on the denotational semantic definition of software assets. Unlike operational methods, they can also be applied to non-executable assets (such as specifications). These methods proceed by checking a semantic relation between the user query and a surrogate of the candidate asset. The surrogate of the software asset can be a complete functional description, a partial functional description, or a signature of the asset. This class of methods is the subject of section 6.

5. Topological methods. The main discriminating feature of topological methods is their goal, which is to identify library assets that minimize some measure of distance to the user query. This feature, in turn, has an impact on the relevance criterion, hence

on the matching criterion. Whether an asset is relevant cannot and need not be decided by considering the query and the candidate asset alone, since the outcome depends on a comparison with other assets. This class of methods is the subject of section 7.

6. Structural methods. The main discriminating feature of structural methods is the nature of the software asset they are dealing with: typically, they do not retrieve executable code, but rather program patterns, which are subsequently instantiated to fit the user's needs. This feature, in turn, has a profound impact on the representation of queries and assets, as well as on the relevance criterion, which deals with the structure of assets and queries rather than their function. This class of methods is the subject of section 8.

For each class of methods that we discuss below, we proceed as follows. First, we present a literature survey, which discusses the main research efforts that fit under the class; in the absence of more meaningful criteria, the order in which we present the methods of each class is the (chronological) order of their publication. In light of the literature survey, we introduce the general characteristics of the class and present a complete profile my means of the table of features given in figure 1. Finally, we discuss how the class of methods fares with respect to the set of criteria we put forth in section 2.2.

3. Information retrieval methods

Before we explore the highly specialized methods of software storage and retrieval, we must recognize that in fact, software assets in general and executable software components in particular can be viewed as documents containing information, in the same way as books or hypertext documents. Consequently, the storage and retrieval of software assets is nothing but a specialized form of information storage and retrieval [Faloutsos 1985; Frakes and Baeza-Yates 1992; Salton and McGill 1983]. To acknowledge this premise, we start our survey by reviewing research efforts that attempt to apply information retrieval technology to the problem of software storage and retrieval. Had the information retrieval technology been adequate to address the issues of managing software libraries, there would have been no need to explore other options – but it has not. Hence we will, in subsequent sections, investigate a wide range of alternative methods. These exploit special features of software assets (e.g., that they are executable, that they conform to precise syntax, that they have clearly defined semantics, etc.) to refine retrieval techniques. The section on descriptive methods can be considered as a link between methods discussed in this section and methods banking on special properties of software.

Information retrieval as a discipline has its tradition in library science. Hence, its basic aim is to better organize the access to the vast body of literature growing dramatically in size, especially since the middle of this century. Thus, "an information retrieval system matches user queries – formal statements of information needs – to

documents stored in a database" [Frakes 1992]. The focus of the stored information is on textual information represented in natural language. The problem of querying an information retrieval system is how to formulate information needs in such a way that the requester gets referred to most of the relevant documents (for the sake of recall) without being distracted by irrelevant documents (for the sake of precision). One of the important design problems of an information retrieval system is therefore, to keep both values as high as possible for well formulated queries and to achieve this at reasonable cost. From this formulation it does appear that information retrieval design is a multi-criterion problem, a premise that is further borne out by studies of combined measures conducted by, e.g., [Van Rijsbergen 1979]. The quality of information retrieval can be achieved by: investing in support structures over the document base, and/or investing in an adequate formulation of the query.

3.1. Literature survey

In this section, we discuss retrieval support based on natural language properties of software assets. As there is quite a range of such assets (from analysis documents to users manuals and comments in executable code) the approaches vary substantially. Nevertheless, they cover just part of the variety of techniques known in information retrieval in general [Frakes and Baeza-Yates 1992].

In [Frakes and Nejmeh 1987a,b], Frakes and Nejmeh apply an information retrieval method to the storage and retrieval of software assets in a software library. They use a strictly free-text approach to the indexing of software assets, which are C programs. Each C module is characterized by a set of single-term indices, which are extracted from the natural language headers; for the sake of uniformity, these indices are extracted automatically by a natural language parser. Because the headers are not subject to any enforceable standard, this method depends critically on a uniform programming discipline, especially on structured commenting standards. Also, because the method does not use a controlled vocabulary, it can only be used by reusers who are familiar with the indexing *habits* of the natural language parser. Both of these premises place limitations on the scope of this method, as we will discuss in section 3.2.

In [Maarek and Berry 1989; Maarek *et al.* 1991], Maarek, Berry and Kaiser discuss an information retrieval system for the storage and retrieval of software components. The method of Maarek *et al.* relies on a natural language description of assets and a natural language representation of queries. Software assets are automatically indexed to ensure uniformity across the library; the indexing process extracts from each asset a set of indices that define its *profile*, and that play the role of *surrogate* in subsequent retrieval operations. For the sake of parsimony, portability and scalability, the authors choose an indexing technique that is based on uncontrolled vocabulary; in order to make up for the loss of precision that stems from this option, they devise a sophisticated mechanism for identifying relevant indices by optimizing the resolving power of lexical affinities in the text. Indexed documents are organized in the library into a browsing hierarchy, using clustering techniques. Retrieval from the library takes place in three

steps, namely: query specification, whereby the reuser formulates a query according to the authorized vocabulary; linear retrieval, whereby the system identifies components that match (to varying extents) the query and ranks them by how closely they match it; browsing, which is invoked whenever linear retrieval fails, and allows the reuser to interact with the system to orient the search for relevant assets. Maarek *et al.* have developed a tool to support their method, called GURU, and have experimented with it on sample queries. They have assessed the performance of GURU in terms of *user effort*, *maintenance effort*, *efficiency* and *retrieval effectiveness*, as measured by precision and recall.

In [Helm and Maarek 1991], Helm and Maarek apply information retrieval considerations for the retrieval of classes from class libraries; here, the additional information provided by inheritance structures is used. The transition between these rather general concepts and software reuse can be found in the work of Maiden and Sutcliffe: their work on domain modeling [Sutcliffe and Maiden 1994] applies to requirements engineering [Maiden and Sutcliffe 1993] as well as to reuse [Maiden and Sutcliffe 1992]. The basis of their work is to establish analogies between situations (specifications) by abstracting from those properties that are due to a specific domain model. This allows them to distinguish between the core of a specification and properties due to its context; in [Maiden *et al.* 1995], they report on experiments to acquire software engineers' mental categorizations.

In [Devanbu *et al.* 1991], Devanbu *et al.* analyze Brooks' assessment of the software development field [Brooks 1987] and argue that *complexity* and *invisibility* are two major obstacles in today's state of the practice in software development. To help overcome these obstacles in the specialized context of their company (AT&T), they develop a specialized system called *Large Software System Information Environment* (LaSSIE) that incorporates a large knowledge base, a semantic retrieval algorithm based on formal inference, and a powerful user interface; LaSSIE is intended to help programmers find useful information about a large software system – specifically, an AT&T scalable PBX product. The knowledge base of LaSSIE is built using a classification based knowledge representation language; the knowledge base serves as a repository of information about the PBX product, as well as an index for retrieving reusable components. It is structured as a taxonomy of nodes that have four possible types: *action*, *object*, *doer*, and *state*; nodes of the taxonomy are linked by traditional *IsA* relations. This structure is further enriched by code-level information, in the form of relationships between nodes; these relationships take values such as *function-calls-function*, *source-file-includes-headerfile*, etc. A LaSSIE query is a description of an action, formulated by filling various frame-like slots; LaSSIE resolves a query by retrieving instances of the knowledge base where the structure of the query occurs with the proper fillers. This method could be viewed as an information retrieval method or a descriptive method. We have chosen to classify it as an information retrieval method because, like such methods, it performs a matching by analyzing natural language descriptions of queries and assets, using artificial intelligence techniques and methods.

The work of Clifton and Li [1995] as well as of Mittermeir and Würfl [1995] can be considered as instances of information retrieval methods. In both cases, *conventional abstractions* are used to describe software. In [Clifton and Li 1995], Clifton and Li use design information as abstraction and propose neural network technology to accomplish the match. In [Mittermeir and Würfl 1995], Mittermeir and Würfl suggest to perform automatic code analysis and propose to match the resulting flow-graphs with control and data flow sketches done by the programmer. Somehow related to these approaches is also the RVM system; this *Reuse View Matcher* is a reuse documentation tool that makes explicit the usage context of an example. The empirical study conducted with RVM focuses on qualitative characterization of expert reuse [Rosson and Carroll 1996] in the context of Smalltalk user interface programming. In this context, we might mention also the work of Lung and Urban [1995]. While the method itself is just a combination of an enumerative hierarchy with a faceted method, the fact that they base their approach on an extensive domain analysis and domain abstractions allows us to consider this work under information retrieval methods.

It is reasonable to consider that hypertext methods are specialized forms of traditional information retrieval methods; they differ by their ability to deal with multimedia documents, and their ability to work across computing sites. The work of Lucarella and Zanzi [1996] fits this characterization. Other hypertext approaches include the works of Isakowski and Kaufmann [1996] and Poulin and Werkman [1995]. In [Poulin and Werkman 1995], Poulin and Werkman describe a Reusable Software Library (RSL) interface and a search tool using Mosaic. The interface, developed for the Loral Federal Systems Group RSL is called the *Federal Reuse Repository* (FRR). The FRR provides three ways to locate a component. A user can either use the hierarchical view arranged by language or library, a subject listing or a keyword search. The hierarchical view allows a search based on the implementation language. If a user decides to search for a component that performs a particular function then he/she will select the FRR, *arranged by Subject* option. The third option consists of a simple keyword search implemented using standard ISINDEX WAIS. The authors also present a technique called *structured abstract* (SA) that allows users to submit queries to the system. SAs include several items such as computer language and component type, domain, function, data, operating system, elements and contact. The user is required to enter the same information in the same order every time he/she wants to search for a component.

ASSET (Asset Source for Software Engineering Technology) [ASSET 1993] is a software library organized by the Advanced Research Projects Agency (ARPA) under the STARS program. An asset in ASSET is defined as either a reusable software component or a document describing some aspects of software reuse or software engineering. In order to submit an asset to the library, the library administrator must produce the following information: name of the component; alternate name; release date; asset size; asset type; distribution code; domain; keyword; reference number; computer language; computer environment; format; author name; abstract; support available; contact person; producer; supplier; supply date. The Asset library provides six ways to search for a component. The first option, search by *domain* returns a list

of assets classified by subject area or application. A search by *collection* is based on a listing of the origin of an asset (CARDS, FREEWARE, SPC, etc.). A search by *Evaluation level* uses a listing of assets by quality evaluation level. The *asset name* option provides an alphabetical listing of assets by title or name. The *asset identifier* option provides a numeric listing by asset unique identifier. *New holdings* corresponds to the listing by date of the assets added to the library. Noting that software reuse is to a large extent reuse of related textual documents, we point here also to the *Kabiria* system described in [Celentano *et al.* 1995].

To conclude, we mention that approaches based on general information retrieval ideas are not confined to the software reuse literature. To give an example from the literature on reuse in the information systems context, we refer to [Castano *et al.* 1996]. An approach based on query generalization, structural and semantic mappings on the goal level is reported in [Massonet and Lamsweerde 1997]. To conclude this line of reasoning and point to future applications of information retrieval technologies in reuse related software engineering, we might refer to Potts' work on naturalistic inquiry [Potts and Newstetter 1997] as requirements elicitation technique. These techniques, like heavy dependence on use-case technology for requirements description, will certainly pose new challenges to information retrieval technologies in software development, demanding answers that cannot yet be given by the approaches discussed in the sections that follow.

3.2. Characterizing information retrieval methods

We review in turn the main attributes of software libraries, as we introduced them in section 2.3, and discuss how information retrieval methods can be characterized in terms of these attributes.

- *Nature of the asset*. By their very nature, information retrieval methods make no restriction on the nature of the asset, since in effect anything that can be characterized by a natural language description can be stored in an information retrieval system. Some projects [Frakes and Nejmeh 1987a,b] have chosen to specialize their method to source code components, but this restriction is not inherent to the method.
- *Scope of the library*. Because they are dependent on the precise interpretation of terminology and on the indexing *habits* of natural language analyzers (when applicable), information retrieval methods are restricted to a very limited scope. Indeed it may be unreasonable to expect personnel to use exactly the same terminology across corporate boundaries, or even across departmental boundaries within the same organization. This is borne out by the fact that operational systems using information retrieval methods are usually confined to a development shop within a corporation, perhaps dedicated to a single (large) product [Devanbu *et al.* 1991].
- *Query representation*. As befits methods that stem from information retrieval technology, information retrieval methods generally accept natural language queries.

Attributes	Characterization
Nature of asset	Generally unrestricted.
Scope of library	Within a project.
Query representation	Generally, natural language.
Asset representation	Natural language, characteristic indices.
Storage structure	Hierarchical.
Navigation scheme	Follows semantic links.
Retrieval goal	Informal inclusive approximate retrieval.
Relevance criterion	Total or partial match of surrogates.
Matching criterion	The same as relevance criterion.

Figure 2. Characterizing information retrieval methods.

Typically the same indexing mechanism that analyzes library assets is used to analyze queries, so as to ensure uniformity of interpretation. In some cases [Devanbu *et al.* 1991] queries are represented in a specialized knowledge representation language.

- *Asset representation*. Assets are generally represented in natural language, although they usually delegate a surrogate to retrieval operations; typically, this surrogate is derived from the source asset by some uniform automatic indexing mechanism [Maarek and Berry 1989; Maarek *et al.* 1991].
- *Storage structure*. Information retrieval methods typically organize their software library into a hierarchy, using semantic links (*IsA*, *Calls*, *Refersto*). While this structure is not critical to the effectiveness of these methods, it is important for their efficiency.
- *Navigation scheme*. The navigation scheme of retrieval operations follows the semantic links that define the storage structure; the navigation scheme of browsing operations is driven by the interactions of the system with the reuser.
- *Retrieval goal*. Because they are relatively informal, information retrieval methods are unable to identify components that are *correct* (in the sense of structured programming) with respect to a query – not to mention that the representation of queries does not even allow us to define correctness, let alone check it. Hence the retrieval goal is a vague notion of relevance, to the effect that the retrieved assets either solve the object of the query or can be modified economically to do so. It is noteworthy that information retrieval methods do not make a final statement on the relevance of any particular asset that is retrieved – rather they depend on post-retrieval analysis to make the final determination.
- *Relevance criterion*. The relevance criterion of information retrieval methods is typically a (total or partial but optimal) match between the query (or a surrogate thereof) and the candidate component (or a surrogate thereof).
- *Matching criterion*. Because the relevance criterion can be checked in a computationally efficient manner, there is no difference between matching criterion and relevance criterion.

This characterization of information retrieval methods is summarized in figure 2.

3.3. Assessing information retrieval methods

In this section we review the evaluation criteria set forth in section 2.2, and discuss how the class of libraries discussed in this section fares with respect to these criteria.

3.3.1. Technical criteria

1. Precision. It is difficult, in principle, to estimate the precision of a method when the very notion of *relevance* has not been clearly defined (see section 3.2). Indeed, the definition of precision (as well as recall) depends on the precise determination of when a retrieved component is considered to be relevant with respect to a query; because, as we discussed in section 3.2, the notion of relevance is not clearly defined, so is the measure of precision. With this qualification in mind, we consider the experimental results proposed by Maarek *et al.* [1991], which provide the following table:

Recall	Precision
0.1	0.86
0.3	0.84
0.5	0.76
0.7	0.58
0.9	0.52

As shown by this table, recall and precision are inversely related and if high recall is aimed at, severe degradations in precision and hence in manual post-selection have to be accepted. Although sophisticated mechanisms can be used to improve the relationship between recall and precision, we give the rate the precision of information retrieval methods as *Medium*.

2. Recall. Maarek *et al.* [1991] report an average recall of 0.88: on average, nine out of ten relevant assets are retrieved, and one out of ten is forgotten. To account for the unavailability of a precise definition of relevance, and for the fact that other information retrieval methods are not as sophisticated as that of Maarek *et al.*, we rate the recall of information retrieval methods as *High*.

3. Coverage ratio. Many of the information retrieval methods have a non trivial storage structure, and use it effectively to perform selective navigation [Maarek *et al.* 1991; Devanbu *et al.* 1991]. We rate the coverage ratio of information retrieval methods as *Low*.

4. Time complexity per match. The indexing step of information retrieval methods is typically linear in the length of the query, perhaps polynomial at worst; the matching of surrogates is typically constant, perhaps linear at worst. Overall, we rate the time complexity of information retrieval methods as *Low*.

5. Logical complexity per match. The indexing step of information retrieval methods, which is applied to submitted queries, involves some non-trivial but fairly straightforward natural language analysis; the matching step is typically at the level of complexity of a compound boolean, with perhaps some non-trivial but straightforward distance computations (lexical affinities, resolving power). Overall, we rate the logical complexity of information retrieval methods as *Medium*.

6. Automation/automation potential. Information retrieval methods rely on a relatively old technology, for which support tools are widely available. However, in order to obtain high performance from such tools, one needs to customize them, parameterize them, and instantiate them with concepts and terms that are specific to the environment where they are used. We rate the automation/automation potential of information retrieval methods as *High*.

3.3.2. Managerial criteria

1. Investment cost. Information retrieval methods depend on concepts that have been used in practice in such traditional disciplines as library science. This has the following impact: because the concepts it involves are fairly straightforward, little personnel training is required; because it can accommodate natural language descriptions of assets and queries, little change is required in an organization's operational procedures; finally, because it relies on an existing technology, the tools that support it can be acquired at little cost. This is borne out by evidence to the effect that most methods presented in this section use existing tools, and do little specific development of their own. Consequently, we rate the investment cost of information retrieval methods as *Very Low*.

2. Operating cost. Maarek *et al.* [1991] have investigated this very question as it pertains to their system, under the header *maintenance effort*. Operating costs include the cost of indexing new assets and adding them to the library; the indexing is performed automatically (hence at little cost) and the insertion of new components can be done incrementally. Kaplan and Maarek [1990] discuss several algorithms for incrementally updating a library of lexical affinity-based indices when inserting, deleting and modifying software assets. Overall, to the extent that the system of Maarek *et al.* is representative of information retrieval methods, we rate the operating cost of information retrieval methods as *Low*.

3. Pervasiveness. Information retrieval methods are used across several academic institutions [Frakes and Nejmeh 1987a,b], research laboratories [Maarek *et al.* 1991] and development laboratories [Devanbu *et al.* 1991]; perhaps because of their low technological threshold, they are fairly pervasive. We rate their pervasiveness as *High*.

4. State of development. Information retrieval methods have been used as industrial products, be it on an experimental basis and in an In-House environment. We rate the state of development of information retrieval methods as *High*.

Technical						Managerial				Human	
Pre-cision	Recall	Cov. Ratio	Time Compl.	Logical Compl.	Auto-mation	Inv. Cost	Oper. Cost	Perva-siveness	State of Develop.	Diff. of Use	Trans-parency
M	H	L	L	M	H	VL	L	H	H	M	H

Figure 3. Summary of assessment: information retrieval methods.

3.3.3. Human criteria

1. Difficulty of use. To the extent that they accommodate natural language queries, and that their concept of operation is straightforward, information retrieval methods are fairly easy to use. On the other hand, to the extent that they feature unreliable precision and recall, hence delegate much of the retrieval/selection/assessment activity to the reuser, they might be tedious to use. Overall, we rate the difficulty of use of information retrieval methods as *Medium*.

2. Transparency. To the extent that they do not involve the user in the search process, information retrieval methods are fairly transparent; on the other hand, to the extent that the user must have some understanding of the selection mechanism of the method (e.g., how they compute lexical affinities, how they navigate through the knowledge base, etc.) in order to make the best interpretation of their outcome, these methods are not perfectly transparent. We rate the transparency of information retrieval methods as *High*.

This assessment of information retrieval methods is summarized in figure 3.

4. Descriptive methods

Descriptive methods proceed by matching a keyword-based query against assets that are represented by (structured) lists of descriptive keywords. A candidate asset is selected whenever the keywords that form its representation match all (or most) of the keywords that form the query. Because of the simplicity of its concept and the convenience of putting it in operation, this family of methods is widely used nowadays. Of course, this scheme is a special (simple) representative of information retrieval in general. However, we discuss it here as a special case due to its pervasiveness in the reuse practice and literature and since it also constitutes a very coherent category.

The relationship between the library asset and its descriptor is established by a human, let's say the library administrator. The administrator is responsible for defining the abstraction of the asset that adequately describes the asset (concisely, faithfully, without redundancy, without information overloading, etc.). This abstraction process is not without pitfalls: we have to assume that the person inserting a component and the person looking for this component have the same frame of reference when working with the reuse library; otherwise, they might refer with different terms to the same concept [Mittermeir 1998]. Thus, the assumption that the library's administrator is responsible for providing a descriptor (as opposed to making the component's provider responsible

for this abstraction) is important from the point of view that this centralization of the description/abstraction process ensures a minimum of consistency among descriptions for conceptually related components included into the library at different times. A list of additional issues to be considered when forming descriptors can be found in [Frakes and Gandel 1990].

Thus, we note a conceptual separation between the library asset and its representation in the library. This separation has the advantage that descriptive methods are universally applicable, irrespective of the nature of the asset to be described. It has the disadvantage though, that even a complete match between descriptors cannot guarantee a match between the items thus described. Nor can we infer from the failure to find a matching descriptor, that there is no matching component.

4.1. Literature survey

The simplest way to describe an asset is to attach to it a (set of) keyword(s). This *keyword based* approach is indeed the archetype of descriptive method. By analogy with what we find in the subject index of a library of books, a software asset is represented by a set of keywords [Matsumoto 1993]. While the state-of-the-art has advanced beyond simple keyword representations, the notion of attaching a set of words to an asset is still in use in modern reuse terminology, where the process of describing components is referred to as *classifying* them [Karlsson 1995]. Due to the difficulty of maintaining consistency and accuracy, pure keyword based approaches are limited to in-house software libraries of limited size.

The approach most widely discussed in the reuse literature is the *faceted approach* introduced by Prieto-Diaz [Prieto-Diaz 1985, 1991; Prieto-Diaz and Freeman 1987]. It traces its roots to library science and extends the simple keyword based approach in so far as an asset is no longer described by an unstructured set of keywords. Rather, a multi-dimensional search space is defined, with each dimension, referred to as *facet*, made up of a set of predefined keywords. An example of such facets might be [Prieto-Diaz 1991]: function, object/item-type, medium, system-type, functional-area, setting. For each of these facets, a set of keywords, its term space, is given (e.g., for function: add, append, close, compare, etc.). Thus, having the domain of a library structured into n facets implies that a query into the search space of this library would be made up of an n-tuple of keywords with the ith keyword drawn from the term space of the ith facet. There is no particular order in the term space and in principle, one might assume that the set of terms defining a given facet is open to grow as demand arises. However, having an open vocabulary for a facet would in the long run lead to inconsistent classification. Thus, the library's administrator is not only responsible for appropriately classifying components but also for defining the term space and for controlling its evolution if the need arises. To broaden the concept, given retrieval tools might allow for boolean operations among terms. Likewise, one might add a thesaurus to better control the issue of synonyms, or one might build term hierarchies to provide for generalizations.

Related to the faceted approach is the notion of the four-dimensional classification cubeproposed in [Mittermeir and Rossak 1987, 1990]. It aims at supporting fine grained search, i.e., at discrimination of components after a set of potentially qualifying candidates have been already identified by standard keyword search or search in the spectrum of given facets. There, a classification of components along properties related to the representation, structure, and semantics of components is given. Thus, component-subcomponent relationships (*PartOf*) and generalization-specialization relationships (*IsA*) can be maintained among components. This additional information matters specifically, if components are to be directly integrated into a platform for development with reuse [Hochmueller 1992].

In [Boerstler 1995], Boerstler proposes with his *feature-oriented classification* a hierarchical refinement of the faceted approach. What he calls a *view-refinement* corresponds essentially to classical facets. As a given component would be classified according to various facets, it can be classified according to the various attributes one might attach to a feature (or, in the authors terms: views of a feature). However, the view categories can be refined, such that a faceted tree structure emerges. When a sufficient level of detail has been obtained, one switches from the view-refinement to *is-refinement*. Here, one out of a choice of conceptually disjunctive attributes is selected to characterize the component. This attribute can be further refined into (disjunctive) sub-categories.

Another extension of the basic concepts of keyword based classification is classification by weighted term spaces or by fuzzy relations [Karlsson 1995]. Basically, these concepts are drawn from the respective information retrieval concepts, applied to software description. So are various elaborate schemes for forming vocabularies of descriptors [Frakes and Gandel 1990].

Certain knowledge based approaches can be considered as another form of elaboration of descriptive methods. The classification of knowledge based approaches in general provides some difficulties, since their degree of formalization is certainly above those of conventional keyword based or faceted descriptive approaches. However, as long as the core semantics are based on the natural-language semantic of strings, it seems fair to discuss them in this context. In addition to conventional knowledge based approaches based on semantic nets or rules, case based reasoning approaches to the reuse problem have emerged recently [Talents *et al.* 1997; Gomez 1997; Gonzalez and Fernandez 1997]. Their merits is on one hand, that they can deal with quite intricate patterns, while on the other hand, one does not need to perform an extensive domain analysis to come up with highly discriminative descriptors.

Supporting the premise that descriptive methods can be used to represent a wide range of assets, Khoshgoftaar *et al.* [1997] discuss a *case library*, whose entries (called *cases*) represent descriptions of software modules along with records of their behavior with respect to faults. Khoshgoftaar *et al.* use this library to help predict the reliability of modules under development by analyzing the past behavior of similar modules stored in the library; the similarity between a given module (submitted as a query) and

a candidate library module is measured by the number of common features that they have in a predefined list of attributes (which act as facets in the descriptive method).

4.2. Characterizing descriptive methods

To characterize descriptive methods in general, we might state:

- *Nature of asset.* Since, paraphrasing Wittgenstein, everything worth describing can be described by words, descriptive methods can be used to describe any type of reusable software asset. In the methods that we have discussed in this section, assets range from software components to reliability histories.
- *Scope of the library.* The scope of the library is heavily dependent on the quality one invests in the definition and standardization of descriptors. With an ad-hoc and open vocabulary, one can hardly transcend the scope of an individual programmer or a small intensely cooperating group. Because the performance of descriptive methods depends heavily on a precise interpretation of terminology, and because this terminology must be shared between reusers and library administrators, it is difficult to envisage using descriptive libraries across organizational boundaries. Within an organization, the quality of classification lexica will be a key factor. Considering the classification problems referred to in [Mili *et al.* 1997], it seems fair to consider the departmental level of the site level as the most realistic scope of descriptive methods.
- *Query representation.* This depends on the particular descriptive method put into operation. With the faceted approach, a query is a vector containing one keyword per facet. Term combinations may be used with other approaches or with extensions of the basic concept. Other approaches, such as the cube proposed by Mittermeir and Rossak [1987, 1990] implicitly assume a kind of guided browsing through the (already limited) set of candidate assets.
- *Asset representation.* As there is no limit on the kind of assets, there is also no particular constraint on their representation. By and large, software assets are represented by a structured set of keywords, which are taken from predefined vocabularies.
- *Storage structure.* Storage structure is not an important factor in the operation of descriptive methods. In the original faceted approach [Prieto-Diaz 1985, 1991; Prieto-Diaz and Freeman 1987], there is no storage structure to speak of: the library is viewed as an unordered, unstructured, set of assets; any structure that there may be between assets is a feature of the database implementation that supports the method, rather than a feature of the method. The method discussed by Hochmüller and Mittermeir [Hochmüller 1992] does use a genericity relationship between assets as a basis for structuring assets in the library.
- *Navigation scheme.* With descriptive approaches that choose the descriptors irrespective of the detailed semantics of the individual assets, navigation consists basically of widening or restricting the query depending on the yield (recall and

Attributes	Characterization
Nature of asset	Any software asset.
Scope of library	Limited by scope of common terminology/frame of reference.
Query representation	One (or more) keywords per facet.
Asset representation	One (or more) keyword per facet.
Storage structure	Indexed resp. facet-based classification.
Navigation scheme	For each facet: linear.
Retrieval goal	Retrieving all assets that pertain to keywords.
Relevance criterion	Keyword matches.
Matching criterion	The same as relevance criterion.

Figure 4. Characterizing descriptive methods.

precision) of the previous query. With approaches supporting also inter-component dependencies [Mittermeir and Rossak 1987; Rossak and Mittermeir 1989], navigation can be made along those lines (is-a and part-of relationships) [Mittermeir and Rossak 1987; Rossak and Mittermeir 1989].

- *Retrieval goal.* The *implicit* goal of descriptive methods is to retrieve all the library assets that deal with the topic defined (by means of keywords) by the query. Because descriptive methods typically perform an exhaustive search of assets [Prieto-Diaz 1985, 1991; Prieto-Diaz and Freeman 1987] or exclude from consideration only assets that are known to be irrelevant [Mittermeir and Rossak 1987; Rossak and Mittermeir 1989], the retrieval goal is the universal quantification of the relevance criterion.
- *Relevance criterion.* An asset is considered to be relevant with respect to a query if and only if it deals with the topic of the query. In the case when assets are executable software components, the relevance criterion may be that the asset solves some problem defined by means of the query, or that the asset can be easily modified to solve the problem in question.
- *Matching criterion.* An asset matches a query if and only if all the keywords of the query match the corresponding entries of the asset. Clearly, there is a wide gap between the relevance criterion and the matching criterion that attempts to implement it. This gap is due to a variety of features, including: the reuser's interpretation of the keywords and the library administrator's may be different; the facets are not sufficiently rich to provide an accurate characterization of the asset; or they are not sufficiently rich to provide a complete characterization of the reuser's intent; etc.

This characterization of descriptive methods is summarized in figure 4.

4.3. Assessing descriptive methods

With respect to the evaluation criteria given in section 2.2, the following assessment of descriptor based methods can be made:

4.3.1. Technical criteria

1. Precision. We have to distinguish precision on two levels. Due to the manual process of classifying components as well as the vague semantics of words drawn from natural language, precision with respect to retrieving (only) correct components cannot be assured. Considering precision on the descriptor level however, this approach raises no serious concerns. Since it is assumed, that a limited and controlled vocabulary of descriptors is used, the match between descriptors is rather of the nature of matching a database query against the contents of the database. Hence, the fuzziness of the information retrieval problem at hand is rather confined to the relationship between the query and the asset. Overall, there are no intrinsic limitations to the precision of descriptive methods: with a sufficient number of facets and a sufficiently large vocabulary on each facet we can obtain arbitrarily high precision. This argument seems also substantiated by the empirical results reported in [Frakes and Pole 1994]. Considering the variations of approaches actually in use, we rate the precision of descriptive methods as *High*.

2. Recall. The remarks just made for precision apply. Good recall of assets requires good and consistent qualification of components. Good recall of relevant descriptors is in general not the problem of these approaches. As with precision, there are no intrinsic limitations to the recall of descriptive methods: with a sufficient number of facets and a sufficiently large vocabulary on each facet we can obtain arbitrarily high recall. Practical realizations however, are far from perfect recall [Frakes and Pole 1994]. Hence, we rate the recall of descriptive methods as *High*.

3. Coverage ratio. Descriptive methods prescribe no particular navigation scheme, hence the default option is an exhaustive scan of all the components in an arbitrary order. This yields a rating of *Very High*.

4. Time complexity per match. The matching condition of descriptive methods consists in the conjunction of elementary conditions, which are in turn simple equalities between two string variables; the number of conjuncts is predefined, and equals the number of facets used in the method. This is a constant time complexity, yielding a rating of *Very Low* for this factor.

5. Logical complexity per match. The matching condition of descriptive methods is a compound boolean expression, yielding (according to the criteria set forth in section 2.2) the rating *Low*.

6. Automation/automation potential. Descriptive methods are very easy to implement and hence eminently automatable. Standard database technology or browser technology suffices to implement small keyword based retrieval mechanisms. Hence, this is the approach most widely used and with most privately made implementations. We rate the automation of these methods as *Very High*. Note, however, that finding appropriate keywords or descriptors is a purely manual task, a task that is sufficiently cumbersome to limit the size of such repositories.

4.3.2. Managerial criteria

1. Investment cost. As we have seen from the above discussion, the quality of the term set (and facets) is the decisive factor of the quality of a retrieval method based on descriptive methods. Further, we have seen that modifications of the term set (let alone facets) will be (prohibitively) costly, since it would require manual reclassification of all assets. Thus, although these approaches seem to be conceptually simple, careful domain studies are to be performed before this approach is put into operation. This becomes evident when establishing a faceted method, but less so when embarking on simple keyword based methods. For this reasons, organizations often unduly shy away from implementing a faceted organization method by looking at the initial effort and the immediately visible complexity of the set-up. In light of this discussion, we rate the investment cost of descriptive methods as *High*.

2. Operating cost. We have to distinguish between incorporating an asset that can be adequately described by the term set already established and extensions of the term set. The former is relatively trivial. It amounts to the effort of classifying the component along the terms given. This activity will best be done by a central authority (repository administrator) in cooperation with the component's developer. Once this is done, physical incorporation is a trivial insert operation. If incorporating a new component imposes an extension of the term set, this operation might be very complex. It should involve reconsidering the complete term set (pertaining to the respective facet) and possibly reclassifying all components under the revised term set. Alternatively, one might construct a meta-structure over the term set. This will be a less expensive operation in the individual evolutionary step, but it will downgrade the discriminative power of the term set in the long run. In light of this discussion, we rate the operating cost of descriptive methods as *High*.

3. Pervasiveness. Because of their low technological threshold, descriptive methods are fairly pervasive in development shops. Surely, the investment cost and operational cost associated to these methods are obstacles to their widespread use, but this cost is applicable only if we insist on excellent retrieval quality (precision, recall, response time, etc.). Empirical studies [Frakes 1991; Frakes and Fox 1993; Frakes and Pole 1994; Tracz 1988] routinely show that simple methods are most widely used in industry. Hence we rate the pervasiveness of descriptive methods as *High*.

4. State of development. Because of the low technological threshold of descriptive methods, the gap between a laboratory prototype and a fully supported industrial product for this method is fairly narrow. With today's technology, it is quite possible to produce fully supported industrial products that implement descriptive methods; hence we rate the state of development of descriptive methods as *Very High*.

4.3.3. Human criteria

1. Difficulty of use. Descriptive methods are fairly easy to use, as they rely on very simple concepts, which are familiar to all reusers; we rate their difficulty of use as

Technical						Managerial				Human	
Pre-cision	Recall	Cov. Ratio	Time Compl.	Logical Compl.	Auto-mation	Inv. Cost	Oper. Cost	Perva-siveness	State of Develop.	Diff. of Use	Trans-parency
H	H	VH	VL	L	VH	H	H	H	H	VL	VH

Figure 5. Summary of assessment: descriptive methods.

Very Low.

2. Transparency. Descriptive methods can be made very transparent; we rate their transparency as *Very High*.

This assessment of descriptive methods is summarized in figure 5.

5. Operational semantics methods

Operational semantics methods recognize that in the context of information retrieval, software components have a discriminating feature that sets them apart from other retrievable assets: they are executable. These methods use the executability of software components as a basis for the selection of candidate assets from a software library.

5.1. Literature survey

In [Podgurski and Pierce 1992, 1993], Podgurski and Pierce make the statistical observation that a software component can be uniquely identified within a large software library on the basis of its behavior on a few randomly selected sample inputs. From this observation, they derive a software library with the following characteristics:

- Components are represented by their executable code. In addition, the description of a component includes a definition of its input space by means of typed variables, as well as a probability distribution on its input space (called the *operational input distribution*), which reflects the likelihood of each input state in a typical execution.
- Queries take the form of an *interface specification*, which defines the desired input space, as well as an *oracle*, which determines whether the behavior of a candidate library component (represented by an input–output pair) is consistent with the *reuser*'s intent.
- Matching proceeds by selecting an input sample from the submitted interface specification; this sample is selected randomly from the input space according to the operational input distribution. All the components that satisfy the oracle for all the selected inputs are retrieved.

Podgurski and Pierce find that the probability that a candidate component be irrelevant to the query despite satisfying the oracle on n randomly selected inputs decreases exponentially with n; hence in practice we can achieve great precision with small

values of n. On the other hand, recall can be controlled by weakening the interpretation of the oracle or by accommodating variable renaming and sub-typing relationships.

In [Hall 1993], Hall discusses a number of weaknesses in the basic behavioral sampling (BS) technique of Podgurski and Pierce [1992], and proposes means to cope with them; this gives raise to what he calls *generalized behavior based retrieval* (GBR).

- BS has low recall because it lacks the flexibility to deal with optional arguments and implicit argument values (as practiced, e.g., by Ada). GBR deals with this contingency by making provisions in its matching conditions for optional inputs; this is all the more pertinent for software reuse that generic software components typically have optional arguments.
- BS's policy of selecting inputs on the basis of a component's operational input distribution is unnecessarily complex, and may in fact adversely affect the retrieval precision. Inputs that occur often in a typical usage are not more likely to discriminate a component than inputs that occur seldom. GBR deals with this issue by letting the reuser select sample inputs.
- BS's policy of executing candidate components may be impractical if the component in question has non-trivial side effects (e.g., deleting files). GBR deals with this issue by using side-effect-free *functional models* of the components at hand rather than the components themselves.

Furthermore, for the sake of recall, Hall [1993] also weakens the relevance criterion from being correct with respect to the reuser's oracle to being useful in developing a programming solution to the reuser's oracle. Finally, at the implementation level, GBR makes provisions for controlling the execution time and the termination of the components that it invokes. Hall runs two experiments to illustrate GBR: an experiment that involves a library of 161 data processing components, and produces an average search time of 40.2 seconds; a smaller library of file manipulation functions under Unix, which produces an average search time of 2.1 seconds. The search times given here reflect the cumulative CPU time that the system spends matching library components against the query; the *actual* time that it takes to perform these queries is, of course, much longer, because it involves an interaction between the reuser and the system (entering sample data, checking outputs, attending to exceptions, etc.).

In [Atkinson and Duke 1994], Atkinson and Duke combine the behavioral sampling idea of Podgurski and Pierce [1992] with the lattice structuring idea of Mili *et al.* [1994] to produce a software library organization for the storage and retrieval of classes, in the sense of object-oriented programming. Atkinson and Duke provide a language-independent abstract model for classes and class behavior, based on relation theory and represented in Z. The behavior of a class is represented by an input/output pair, where the input is a sequence of incoming messages and the output is the corresponding sequence of responses. From this formal model they derive a partial ordering on class behaviors (which reflects that one classe's behavior subsumes another's), and discuss the lattice properties of this ordering; they find that any two behaviors have a meet, and that two behaviors have a join only under some *compatibility* condition.

The lattice structure so exhibited is used as the basis of a retrieval algorithm, whereby for a given query, we select library classes that maximize similar behavior with the query and minimize distinct behavior from the query.

In [Chou *et al.* 1996], Chou *et al.* present a behavior-based storage and retrieval technique for Object-Oriented software components. They argue that the payoff of software reuse increases with the size of the reused assets, and advocate a flexible mechanism that allows the retrieval of components of arbitrary size and complexity; in an Object-Oriented context, this means that the reusable unit is a class or a combination of classes. Also, Chou *et al.* recognize that reuse is most effective when it is applied in the earlier phases of the life-cycle, hence focus on *specification* reuse. Following Booch [1994], Chou *et al.* specify the behavior of a class by means of state transition diagrams; also, they classify library classes (and larger systems) by means of a semantic network. Candidate classes or systems are matched against queries by comparing their behaviors: the proposed technique selects those entries that maximize similarity with the query; similarity is measured as the ratio between the cardinality of common behaviors over the overall cardinality of behaviors (Jaccard's coefficient [Van Rijsbergen 1979]). Chou *et al.* produce an automated tool that supports their method; it includes a specification editor, a specification classifier, a repository manager and a specification retriever.

In [Park and Bai 1997], Park and Bais introduce a version of behavioral sampling where input samples are neither statistically selected (as in [Podgurski and Pierce 1992]) nor user-selected (as in [Hall 1993]): rather they are selected on the basis of an inductive argument on the input domain. The system of Park and Bais attempts to recognize an inductive structure in the input domain, then selects input samples in such a way as to test candidate components on the base case and inductive cases. Because in practice, most inductive arguments adhere to simple patterns (induction on the size of an array, on the structure of a binary tree, etc.) it is not difficult to define an inductive structure on the input domain by analyzing its data structures; Park and Bais [1997] discuss an automated system for input sample generation.

5.2. Characterizing operational methods

We review in turn the main attributes of software libraries, as we introduced them in section 2.3, and discuss how operational semantics methods can be characterized in terms of these attributes.

- *Nature of the asset*. By their very definition, operational semantics methods depend on the ability to execute assets; hence by their very definition, these methods are limited to executable assets. The GBR (General Behavioral Retrieval) method [Hall 1993] could in principle lift this requirement, to the extent that it represents assets by their *functional models* – but to our knowledge this method is applied to executable assets, nevertheless.
- *Scope of the library*. To the best of our knowledge, all the software libraries that are discussed in this section are small scale prototypes – although with 161 components,

the library that Hall [1993] uses for his experiments is sufficiently large to be a project-wide library. Perhaps most important is the observation that the knowledge required to use a behavioral based software library is fairly general, hence this library organization could conceivably be used on a wide scale (organization-wide, inter-organization).

- *Query representation*. In operational semantics methods, a query is made up of a sample of input data (under some form) and an *oracle*, i.e., an agent that can determine whether an observed behavior is consistent with respect to a specification of the user's intent. For simple functional modules [Podgurski and Pierce 1992; Hall 1993], traditional specifications (by means of functions, relations, pre/post conditions) are adequate. For classes (in the sense of object-oriented programming), Atkinson and Duke [1994] propose a relational model between input sequences and output sequences, and Chou *et al.* [1996] propose a state transition model. For behavioral sampling methods whose input data is generated automatically [Park and Bai 1997; Podgurski and Pierce 1992], no other information is required. In Hall's GBR method [Hall 1993], the query representation further includes user supplied input samples; and in Atkinson and Duke's method [Atkinson and Duke 1994], the user supplies a sequence of input stimuli.
- *Asset representation*. By the very definition of behavior based methods, assets are represented by means of executable source code segments, although the GBR method also maintains a functional model of the asset as part of the representation.
- *Storage structure*. The methods discussed under this section do not mention any particular storage structure, hence assume presumably that the library is organized as a flat list of components. While storage structure is not crucial to behavioral sampling methods, it could nevertheless improve their performance. We have identified three possible structures: by an *equivalence relation on signatures*, whereby all library components that have the same signature are stored together; by an *ordering relation on signatures*, whereby a component is greater than another if it has a superset of its parameters [Hall 1993]; by an *ordering relation on behaviors*, whereby a component is greater than another if its behavior subsumes the other [Atkinson and Duke 1994; Chou *et al.* 1996].
- *Navigation scheme*. Attempting to match a component against a query does not make sense unless the component and the query have the same signature (modulo some equivalence or ordering relations). Hence a natural navigation scheme consists in scanning the library twice [Podgurski and Pierce 1992]: once to isolate components that have the same signature as the query; and a second time to match the behavior of the selected components against the query. Atkinson and Duke [1994] further refine this navigation scheme by dividing the behavioral matching in two steps: first they identify those components that have most in common with the query, then they choose among the selected components those that have least apart from the query (or vice-versa).

Attributes	Characterization
Nature of asset	Executable components.
Scope of library	Typically, within product line (application domain).
Query representation	Oracle, and sample input data.
Asset representation	Executable code.
Storage structure	Typically flat.
Navigation scheme	Typically exhaustive.
Retrieval goal	Retrieving all correct components.
Relevance criterion	Correct behavior on sample data.
Matching criterion	Correct behavior on sample data.

Figure 6. Characterizing operational semantic methods.

- *Retrieval goal*. Podgurski and Pierce [1992], Hall [1993] and Park and Bai [1997] all aim to retrieve all the library components that are correct with respect to the specification provided in the query. Atkinson and Duke [1994] and Chou *et al.* [1996] define a more ambitious goal, which is to identify components that approximate the query – with the understanding that if a library component actually satisfies the query then it will certainly be selected among those components that are found to approximate the query.
- *Relevance criterion*. In the scan that selects components on the basis of their signature, the relevance criterion is that the query and the candidate component have the same signature (modulo some equivalence or some ordering relation). In the behavioral matching phase of Behavioral Sampling [Podgurski and Pierce 1992] and Generalized Behavioral Retrieval [Hall 1993], the relevance criterion is correctness of the component under consideration against the specification part of the query. In the object-oriented behavioral retrieval method [Atkinson and Duke 1994; Chou *et al.* 1996], the relevance criterion is (depending on the strategy and the phase within the strategy) to maximize common behavior or to minimize discriminating behavior between the query and the candidate component.
- *Matching criterion*. The matching criterion of all the behavioral methods is the same: it is to compare the behavior of the candidate component on an input sample against the expected behavior as prescribed by the user.

Figure 6 summarizes the discussions of this section, by assigning values to each of the attributes put forth in section 2.3 on the basis of the foregoing discussion.

5.3. Assessing operational methods

In this section we review the evaluation criteria set forth in section 2.2, and discuss how the class of libraries discussed in this section fares with respect to these criteria.

5.3.1. Technical criteria

1. Precision. In principle, the precision of behavioral sampling methods is not ensured, since a component may well behave as expected on all the input samples yet

fail to satisfy the prescribed requirements on other inputs. But there is both analytical evidence [Podgurski and Pierce 1992] and statistical/experimental evidence [Hall 1993; Podgurski and Pierce 1992] to the effect that this family of methods offers good precision in practice. Furthermore, the precision can be controlled to arbitrary levels by increasing the size of the input sample. Because the precision of behavioral methods can be controlled arbitrarily, and because it happens to converge very quickly to 1, we rate the precision of behavioral methods as *Very High*.

2. Recall. Because the matching criterion is a necessary condition to the relevance criterion (although it is not a sufficient condition), it causes no loss of recall. But operational semantics methods may still experience a loss of recall in the signature matching step: if the equivalence relation between signatures (that capture the property that two signatures are equivalent, modulo renaming and permutation, for example) and the ordering relation between signatures (that capture the property that one signature is a subtype of another) are not sufficiently flexible, some relevant candidate components may well be excluded from further consideration. We rate the recall of behavioral methods as *High*.

3. Coverage ratio. The signature matching step of operational semantics methods has a coverage ratio of one; subsequent steps (which are potentially more complex and more time consuming) have a smaller coverage ratio, since they only scan the components that have been selected in the first step. Because all the entries of the library are considered, and because only a selection of them are inspected carefully, a fair rating should be a middle ground between VH and M; we choose the rating *High*.

4. Time complexity per match. Interestingly, this factor is not under the control of the method, since it involves executing candidate components – hence is driven by the time complexity of the candidate component. This factor can be controlled somewhat for methods where the user selects input samples [Atkinson and Duke 1994; Hall 1993], but is totally uncontrolled for methods where the input samples are automatically generated [Park and Bai 1997; Podgurski and Pierce 1992]. We rate this criterion as *Medium*, to account for the variance in execution complexity of the various candidates.

5. Logical complexity. Whereas time complexity depends on library components, logical complexity depends primarily on the submitted query: it could be as simple as checking the produced output against the expected output, or it could be an arbitrarily complex property of the output. To account for the variance in logical complexity of the produced output, we rate the logical complexity of behavioral methods as *Medium*.

6. Automation/automation potential. The steps involved in the behavioral method are fairly simple, and eminently automatable; in fact most implementations of the behavioral method are at least partially automated. We rate the automation/automation potential of behavioral methods as *Very High*.

5.3.2. Managerial criteria

1. Investment cost. It takes fairly little effort to set up a behavioral sampling based library, besides that of collecting assets. In particular, because library assets are scanned exhaustively, no classification effort is required to store them; in addition, to the extent that the matching condition is merely equality between the sample output and the expected output (and only to this extent), no complex inference mechanism is required. We rate the investment cost of behavioral methods as *Low*.

2. Operating cost. Storage costs are fairly low, because behavioral methods require no special classification of components; on the other hand, retrieval costs vary from component to component, and vary according to the sample data as they involve executing the components. Overall, we rate the operating costs of behavioral methods as *Medium*.

3. Pervasiveness. Originally designed at Case Western Reserve University, this method has been subsequently extended and investigated in several other laboratories, including Lucent Technologies, the University of Windsor, the University of Queensland, and the University of Seoul. While limited to research laboratories (as opposed to development shops), operational methods are reasonably pervasive; we rate their pervasiveness as *Medium*.

4. State of development. Most implementations of operational methods are laboratory prototypes, and some show characteristics of experimental products. We rate the state of development of operational methods as *Medium*.

5.3.3. Human criteria

1. Difficulty of use. In order to use an operational semantics library, a reuser must produce a query under the form of a sample of inputs and their corresponding outputs or a set of inputs and an assertion that outputs must satisfy. We rate the difficulty of use of operational semantics methods as *Low*.

2. Transparency. The user does not have to understand much about operational methods, and in fact there is little to understand because they are fairly straightforward. We rate the transparency of operational methods as *Very High*.

This assessment of operational methods is summarized in figure 7.

Technical						Managerial				Human	
Precision	Recall	Cov. Ratio	Time Compl.	Logical Compl.	Automation	Inv. Cost	Oper. Cost	Pervasiveness	State of Develop.	Diff. of Use	Transparency
VH	H	H	M	M	VH	L	M	M	M	L	VH

Figure 7. Summary of assessment: operational methods.

6. Denotational semantics methods

This section covers software libraries whose operation depends on the denotational semantic definition of software assets. Two types of software assets have been dealt with by this family of software libraries: executable software components, and requirements specifications. Also, assets have been represented at various levels of abstraction, including: functional signatures, functional abstractions, and requirements frameworks. It is a matter of some debate whether specification-based software libraries and signature-based software libraries can be considered as members of the same class or separated into two distinct classes. We have chosen the former viewpoint, on the basis that signature-based retrieval is but a special case of specification-based signature: a functional specification can be represented by a binary relation of the form

$$\{(x, y) \mid x \in X \wedge p(x, y) \wedge y \in Y\},$$

where X is the input domain, Y is the output domain (range) and $p(x, y)$ captures the input–output relation we are interested in. In the special case where predicate $p(x, y)$ is taken to be **true**, we get

$$\{(x, y) \mid x \in X \wedge y \in Y\},$$

which is nothing but a signature definition. Hence, even though these classes of methods have distinct histories (have followed distinct development paths), we present them in the same section. By its very definition, functional matching offers a better precision than signature matching, although at the expense of greater time complexity and greater logical complexity per match.

6.1. Literature survey

In [Perry 1989; Perry and Popovich 1993], Perry and Popovich introduce a prototype of a software library, where software assets (in the form of executable components) are represented by predicates that define their main functional features and interface characteristics. The software library, named *Inquire*, is based on a specification-based software development environment, named *Inscape* [Perry 1989]. Perry and Popovich identify four aspects in the reuse of software systems from building blocks: *conceptualization*, *retrieval*, *selection*, and *use*; rather than focus on assisting the user with the retrieval aspect, the Inquire prototype aims at helping with conceptualization, thereby producing better selection and use – at the expense of some loss of retrieval precision. Inquire has elaborate functions to perform browsing and retrieval, and has special purpose facilities to retrieve software components that implement operations, data objects, and modules; furthermore, to make up for loss of precision, the retrieval function orders retrieved assets according to a user-selected criterion. Ordering criteria are based on software engineering principles, and are intended to present the most desirable candidates first.

In [Rittri 1989], Rittri proposes a method for software component storage and retrieval that applies to modules written in a functional language and is based on

signature matching. The matching criterion is defined using polymorphic type systems and provides independence of the order of components in a type. In [Rittri 1992], Rittri improves the recall of his method by weakening the matching condition using type isomorphisms. Runciman and Toyn [1989] propose a similar solution, which uses polymorphic type systems to define criteria of signature matching in the context of functional programming, and provides independence of the number of arguments; this latter feature is intended to preserve (avoid the loss of) recall in cases when the signatures differ by syntactic details. In [Rollins and Wing 1991], Rollins and Wing present a software library of ML components whose specifications are written in λ-Prolog following a Larch-like [Guttag *et al.* 1985] two tiered approach; they use λ-Prolog's inference capability to automate component retrieval.

In [Gaudel and Moineau 1988; Moineau and Gaudel 1991], Moineau and Gaudel introduce a *theory of software reusability* on the basis of algebraic specifications of software components (more specifically, components that implement abstract data types). Specifications are represented in an OBJ-like language (PLUSS), by describing their signature and axioms which define the interactions between their methods. They define an ordering relation between specifications, the *reusability* ordering, which expresses that some specification can be enriched by the addition of some features to be *equivalent* to another. By varying how two specifications can be considered equivalent, and how some specification can be enriched, they obtain a wide range of matching criteria between specifications. Their criteria make provisions for method renaming and for relaxed equivalence between specifications. In [Moineau and Gaudel 1991], Moineau illustrates the proposed theory by means of an operational system for component retrieval, called *ReuSig*, which deals exclusively with signature matching aspects. A specialized version of this system, which deals with Ada components, is discussed in [Badaro and Moineau 1991] under the name *ROSE-Ada*.

In [Cheng and Jeng 1992a], Cheng and Jeng discuss an organization of a software library that is based on formal specifications of components and queries. Their software library is organized into a two-layered hierarchy by means of a clustering algorithm, where the top layer places together related software components; the purpose of the two-level hierarchy is to apply a two step retrieval process, whereby the first step performs a coarse-grained search to identify a cluster, and the second step performs a fine-grained search on the selected cluster. Cheng and Jeng extend their work in [1992b] by investigating matching criteria that attempt to minimize measures of distance between the query at hand and candidate library components, and they extend it in [Jeng and Cheng 1995] by defining matching criteria between components, and between methods; these matching criteria make provisions for sub-typing, variable renaming, and parameter permutation. Chen and Cheng [1997] extend this work further to deal with software development at the architectural level.

In [Boudriga *et al.* 1992], Boudriga *et al.* discuss the design of a software library based on relational specifications of components and queries, and on an ordering of library components by a refinement ordering relation. The matching condition is defined as the correctness of the component with respect to the query, and the ordering between

library components is used to guide the search process. An experimental prototype of this design is given by Mili *et al.* [1994]; it uses a theorem prover[1], and handles two kinds of retrieval operation, namely exact retrieval (which seeks components that are provably correct) and approximate retrieval (which seeks components that approximate the query). Jilani *et al.* [1997b] extend this work by using measures of functional distance between relational specifications [Mili 1996] to perform approximate retrieval: for every measure of distance, say δ, and for a given query Q, the method identifies all the library components C that minimize the distance $\delta(Q, C)$.

In [Steigerwald 1992], Steigerwald discusses a system that was developed at the Naval Postgraduate School under the name *Computer Aided Prototyping System* (CAPS). The system is geared towards prototyping real-time embedded systems, and includes an execution support system, a syntax directed editor, a software base with an embedded rewrite system and an engineering database management system with an embedded design management system. Queries are formulated at an abstract level using a special purpose specification language augmented with OBJ and the search mechanism relies on syntactic and semantic criteria. The semantic step of the retrieval procedure uses a method called query by consistency, which relies on an OBJ representation of library components and exploits the formal semantics of OBJ to identify a match between the query and candidate components.

In [Zaremski and Wing 1993, 1995a], Zaremski and Wing discuss signature matching as a mechanism for retrieving software components from a software library. Queries and library components are represented by their signatures, and a hierarchy of matching criteria is defined and discussed. The basic matching condition provides for equality between the two signatures, modulo variable renaming and parameter ordering. A number of weaker matching conditions are obtained by relaxing the exact match: by replacing equality with a sub-typing relationship, Zaremski and Wing obtain a *generalized match* (if the type of the component is more general than the type of the query) or a *specialized match* (if the type of the query is more general than the type of the component); also, by applying uncurrying and currying transformations to the query and the component, they can match functions that do not have the same number of parameters. All these relaxations can be combined to produce a wide range of matching criteria. Zaremski and Wing extend their work on signature matching by investigating specification matching in [Zaremski and Wing 1995b]. They represent queries and components by (precondition, postcondition) pairs, and define a general matching criterion under the form

$$\mathit{match}(S, Q) = (Q_{\mathit{pre}} R_1 S_{\mathit{pre}}) R_2 (S_{\mathit{post}} R_3 Q_{\mathit{post}}),$$

where S is the library component, specified by $(S_{\mathit{pre}}, S_{\mathit{post}})$ and Q is the user query, specified by $(Q_{\mathit{pre}}, Q_{\mathit{post}})$. By varying relations R_1, R_2 and R_3 (which are logical connectives), they obtain a hierarchy of matching criteria.

[1] Otter, ©Argonne National Laboratory.

In [Penix and Alexander 1995], Penix and Alexander advocate a formal specification based, domain theory oriented, approach to software component retrieval; they argue that traditional specification matching solutions do not scale up easily, due to the unpredictability of theorem proving, and that a domain theory oriented approach minimizes the impact of theorem proving by creating a domain specific knowledge base. Specifically, a domain is defined by a set of formally specified (or specifiable) features, as well as a body of knowledge that pertains to these features. Arbitrary specifications in this application domain can be formulated by means of these features, so that the match between a query and an available component can take place at a fairly abstract (and computationally efficient) level. They extend their work in [Penix and Alexander 1996] by considering a variety of matching criteria and refining the definition of features. Like Mili *et al.* [Boudriga *et al.* 1992a; Jilani *et al.* 1997a,b; Mili *et al.* 1994], Penix and Alexander make a clear distinction between *exact retrieval* and *approximate retrieval* (rather than to consider the former as a happy coincidence of the latter), and make separate provisions for them; also, like Zaremski and Wing [1993, 1995a,b], Penix and Alexander use a hierarchy of matching criteria and specify components in Larch. Penix *et al.* further extend this work in [Penix *et al.* 1997] by considering software assets that are represented at the architectural level.

Recognizing that most specification matching algorithms fail to perform satisfactorily in practice, due to the bottleneck of theorem proving, Fischer *et al.* [1995] propose a stepwise filtering procedure that proceeds in three steps:

1. Signature matching.
2. Model checking.
3. Theorem proving.

The idea of this approach is, of course, to reduce the search space as the retrieval algorithm gets more and more complex (and less and less likely to converge for large spaces): the first steps attempt to reduce the search space without affecting recall; the later steps attempt to improve precision by applying strong matching criteria to the remaining pool of candidates. Fischer *et al.* apply their approach to a library of components specified in VDM: they find good retrieval precision but poor recall, due primarily to the strong signature matching criterion. In [Schumann and Fischer 1997], Schumann and Fischer experiment with this three-step process to perform large-scale experiments of retrieval of software components in large libraries; the emphasis of their experiment is on providing good response time of the automated inference process, which they achieve at the expense of soundness.

In [Liver 1996, 1997], Liver introduces a method called ZD that is based on FR [Allemang 1990, 1991] and RESOLVE [Ogden *et al.* 1994]. It is targeted towards the description of design patterns as well as modules (and statements). It is a reasoning framework based on the description of structural, behavioral, and functional properties of an asset (called device). The semantics of an asset are described in terms of pre- and post-conditions expressed in Hoare logic [Hoare 1969].

While all software libraries discussed so far deal with executable software components, the library of Massonet and Van Lamsweerde [1997] contains requirements specifications; specifically, it contains requirements frameworks, where a requirements framework is a collection of related requirements concepts. The library is organized as a structured collection of *cases*, where each case is a domain specific aggregation of interrelated goals, constraints, agents, entities, relationships, events, actions, and scenarios; cases are ordered by the *IsA* relationship, which confers the library its hierarchical structure. The retrieval problem from such a library is formulated as follows: given a partially elaborated specification, and given that some parts of the specification have been identified as missing, find candidate library cases that may be used instead of the missing parts. The matching criterion between the query specification (identified as the missing part) and candidate library assets is essentially an analogical mapping (i.e., the query and the component must be found to be analogous). The analogy is subject to a set of constraints, such as: identity of meta-levels (query and candidate must be on the same meta level); identity of arity; role preservation (relationships and predicates can be matched only if their arguments can be matched pairwise). The search for relevant cases is carried out in three main steps: first a structural match is sought between declarations; this match is then refined through semantic matching on case assertions; finally the analogy is extended to ensure its validity.

To the extent that they can be considered as *software assets*, theorem proofs can be considered as possible entries of software libraries. Ihrig and Kambhampati [1995] and Kolbe and Walther [1995] discuss organizations of libraries of proofs.

6.2. Characterizing denotational methods

We review in turn the main attributes of software libraries, as we introduced them in section 2.3, and discuss how the denotational semantic methods can be characterized in terms of these attributes.

- *Nature of the asset.* Generally, the software assets that denotational semantic methods deal with are executable software components. Most of the work on signature matching and much of the work on functional matching deals with software components written in functional programming languages (generally ML), because functional programming languages exhibit the signature of a function in an explicit manner, at the header declaration; also, unlike block structured languages, functional languages do not normally allow variables to be passed implicitly (via block based scoping rules). Other forms of assets are also considered: Gaudel and Moineau [1988] deal exclusively with software modules that define abstract data types; Penix and Alexander [1995, 1996], Perry and Popovich [Perry 1989; Perry and Popovich 1993] and Zaremski and Wing [1995b] discuss the storage and retrieval of software modules, in addition to simple input/output functions; Mili *et al.* [Boudriga *et al.* 1992a; Mili *et al.* 1994; Jilani *et al.* 1997b] do not make a special case for software modules, but that is only because software modules, like software functions, can be specified by relations [Boudriga *et al.* 1992b] – hence

are not given special consideration. Other forms of assets include specification frameworks [Massonet and Lamsweerde 1997] and proofs [Ihrig and Kambhampati 1995; Kolbe and Walther 1995].

- *Scope of the library*. Unless they are geared towards a specific application domain [Penix and Alexander 1995, 1996], denotational methods do not have inherent scope limitations.
- *Query representation*. Retrieval methods that are based on signature matching represent queries by ML-like signatures: a signature specifies the sequence of types of the inputs and the outputs; each type is in turn specified by means of elementary types and type constructors. Retrieval methods that are based on functional matching represent queries by a space specification and a function specification or, in the case of [Gaudel and Moineau 1988; Moineau and Gaudel 1991], and Penix and Alexander [Penix and Alexander 1995, 1996], by algebraic ADT specifications. Space specifications are represented by Pascal-like variable declarations and function specifications are represented either by pre-/post-conditions or by relations. Except for Penix and Alexander, who structure query specifications as aggregates of sub-specifications (called *features*) all denotational semantic approaches represent queries in a monolithic manner. When the library assets are requirements frameworks, the query is represented by the definition of a missing requirement; and when the asset is a proof, the query is represented by a concrete theorem.
- *Asset representations*. When software assets are executable software components, the asset and the query are represented in the same manner. Requirements frameworks are represented by *cases* [Massonet and Lamsweerde 1997], which are aggregations of interrelated objects, actions, agents, goals and constraints; and proofs are represented by *proof shells* [Kolbe and Walther 1995], which are schematic logical formulas.
- *Storage structure*. Generally, denotational software libraries have no structure: their assets are arranged in a flat list. Exceptions include the library of requirements frameworks [Massonet and Lamsweerde 1997], which is ordered by the *IsA* relation, the two-tiered library of Cheng and Jeng [1992a,b], which is clustered by an equivalence relation, and the library of Mili *et al.* [Boudriga *et al.* 1992a; Jilani *et al.* 1997a,b; Mili *et al.* 1994], which is structured by the refinement ordering.
- *Navigation scheme*. The libraries that are not structured are navigated by means of a brute force traversal, which visits in turn all the components, in a predetermined but insignificant order. The library of requirements frameworks [Massonet and Lamsweerde 1997] is traversed following inheritance relations; the two tiered library of Cheng and Jeng is traversed in a two step process, whereby the relevant cluster is selected following a coarse-grained search, then the selected components are traversed in an arbitrary order. The refinement-based library of Mili *et al.* [Boudriga *et al.* 1992a; Jilani *et al.* 1997a,b; Mili *et al.* 1994] is traversed following refinement links between specifications. Fischer, Kievernagel and Snelting [Fischer *et al.* 1995]

advocate a three-pass navigation scheme, whereby the set of candidate components is progressively reduced; in each pass, components are visited in an arbitrary order.

- *Retrieval goal*. Paradoxically, not all methods have a clearly defined retrieval goal. One may consider that in the absence of an explicitly declared goal, the implicit goal is that retrieved components be correct with respect to the query, but this goal is not always satisfied: signature matching does not, e.g., ensure correctness; also, by focusing too much on matching criteria, some methods lose track of their retrieval goal. Penix and Alexander [1995, 1996], and Mili *et al.* [Boudriga *et al.* 1992a; Mili *et al.* 1994; Jiliani *et al.* 1997b], define two distinct retrieval goals: *exact retrieval*, whereby the intent is to extract components that are correct with respect to the query; and *approximate retrieval*, whereby the intent is to extract components that minimize some predefined measure of functional distance to the query.
- *Relevance criterion.* This criterion answers the question: under what condition do we consider that an asset C is relevant with respect to a query Q? This is distinct from *retrieval goal* in the sense that retrieval goal is a library-wide criterion, and is distinct from *matching criterion* in the sense that the matching criterion is an implementation of the relevance criterion. Here again, by focusing too much on matching conditions, methods fail to clearly define relevance criteria. Zaremski and Wing [1995b] offer a number of possible relevance criteria, such as: *satisfies*, *meets*, *is equivalent to*, or *interacts properly with*; they do not show, however, whether the matching conditions provided subsequently implement these relevance criteria. Massonet and Van Lamsweerde [1997] define their relevance criterion as *analogy* between cases and queries. Mili *et al.* [Boudriga *et al.* 1992a; Jilani *et al.* 1997a,b; Mili *et al.* 1994] find that for exact retrieval the relevance criterion is correctness and for approximate retrieval, the relevance criterion is that the distance to the query is minimal; note that this latter criterion is not binary, i.e., it does not involve the query and the component exclusively, but also involves the other components of the library as well.
- *Matching criterion.* This criterion is an implementation of the relevance criterion; it could be different from it, in case practical considerations preclude applying the relevance criterion. For example, in [Boudriga *et al.* 1992a; Jilani *et al.* 1997a,b; Mili *et al.* 1994], components are attached to specifications, and the matching condition compares the query against the specification to which a candidate component is attached rather than to the component itself; this results in a loss of recall, since the component may refine the query, but the specification to which it is attached fails to refine it. In most libraries we have discussed in this section, matching criteria attempt to establish the correctness of candidate components with respect to the query, although many matching conditions could not be traced to this criterion; some matching criteria attempt to minimize measures of distance or maximize measures of similarity.

Figure 8 summarizes our discussions in tabular form.

Attributes	Characterization
Nature of asset	Executable components, specification frameworks, proofs.
Scope of library	Typically, within product line (application domain).
Query representation	Signature specification, functional specification.
Asset representation	Signature specification, functional specification.
Storage structure	Typically flat. Some hierarchical; some partitioned.
Navigation scheme	Typically exhaustive. Sometimes selective.
Retrieval goal	Retrieving all correct components.
Relevance criterion	Functional matching: functional equivalence or refinement.
	Signature matching: data equivalence or refinement.
Matching criterion	Weak (efficient) version of the relevance criterion.

Figure 8. Characterizing denotational semantic methods.

6.3. Assessing denotational methods

In this section we review the evaluation criteria set forth in section 2.2, and discuss how denotational methods fare with respect to these criteria.

6.3.1. Technical criteria

1. Precision. Whenever a retrieval goal or a relevance criterion is not defined, it is difficult to define the *precision* of a retrieval method. By default, we consider that signature matching algorithms attempt to identify all components whose signature (data-) refines that of the query (modulo renaming, currying, uncurrying and permutations), and functional matching algorithms attempt to identify all components that are correct with respect to the query. With this qualification, the algorithms we have discussed produce generally good precision; whenever there is a loss of precision, it stems from the matching condition being insufficient to ensure correctness. It is fair to acknowledge, however, that denotational semantics methods – especially those that use functional matching, have the potential to provide the best possible precision. While this potential is not always fulfilled in practice, due to implementation tradeoffs, it is there nevertheless. Hence we rate the precision of denotational methods as *Very High*.

2. Recall. For algorithms whose navigation scheme is a systematic scan of all the component, the only possible loss of recall stems from the matching condition being unnecessary for correctness. For algorithms such as those of Massenet *et al.* [Massonet and Lamsweerde 1997] and Mili *et al.* [Boudriga *et al.* 1992a; Jilani *et al.* 1997a,b; Mili *et al.* 1994] whose navigation scheme visits components selectively, there may be a loss of recall due to improper routing (nodes are excluded without being irrelevant). For retrieval algorithms such as Gaudel and Moineau [Gaudel and Moineau 1988; Moineau and Gaudel 1991] and Mili *et al.* [Boudriga *et al.* 1992a; Jilani *et al.* 1997a,b; Mili *et al.* 1994], which involve comparing the query to surrogates (specifications) rather than directly to candidate components, there is an additional source of loss of recall: the query may well be refined by the component without being refined by the component's surrogate – hence fail to be retrieved while it is relevant; this loss of recall can, however, be controlled arbitrarily by making the surrogate arbitrarily close to the

component. On balance, we rate the recall of denotational methods as *High*, to account for the fact that it varies widely from method to method, and from implementation to implementation, but can be controlled to arbitrarily high levels.

3. Coverage ratio. Most denotational methods have a trivial coverage ratio of 1: all components are visited for all queries. The only exceptions are: the algorithm of Cheng and Jeng [1992a,b] which focuses on components of the relevant cluster; the method of Massenet and Lamsweerde [1997], which follows *IsA* relationships; and the method of Mili *et al.* [Boudriga *et al.* 1992a; Jilani *et al.* 1997a,b; Mili *et al.* 1994], which follows refinement relationships. Overall, we rate the coverage ratio of denotational methods as *High*.

4. Time complexity per match. Signature matching conditions are fairly simple and have little time complexity, provided they keep the combinatorics of variable renaming and parameter permutation under control (which is not necessarily trivial if one wants to maintain good recall). Functional matching conditions are typically non-trivial logical theorems, whose proof can be quite time consuming; in fact the time it takes to prove theorems generated by these matching conditions is usually the main bottleneck to their time performance. Given the notorious unpredictability of general purpose theorem provers, denotational methods which depend on theorem proving can exhibit very high time complexity. Hence we rate the time complexity of denotational methods as *Very High*.

5. Logical complexity per match. Signature matching conditions are fairly straightforward (including the combinatorial aspects, which, while they are operationally time consuming, are nevertheless easy to model); functional matching conditions vary from simple first order logic theorems [Zaremski and Wing 1995b] to complex formulas of algebraic structures [Gaudel and Moineau 1988]. Overall, by comparison with other simple methods, denotational methods exhibit *Very High* logical complexity, even though their logical complexity is not as pronounced as their time complexity.

6. Automation/automation potential. Signature matching algorithms are fairly automatable, given that the combinatorics of variable renaming and parameter permutations are kept under control. Functional matching algorithms are automatable in principle, although their automation is limited by theorem proving technology – whose performance under general conditions is unpredictable at best. We rate the automation of denotational of denotational methods as *Medium*.

6.3.2. Managerial criteria

1. Investment cost. Most of the methods in this class require a non-trivial up-front investment in terms of programming manpower. We rate the investment cost of denotational methods as *High*.

2. Operating cost. Two cost factors contribute to this term: the cost of maintaining a software library (essentially systems work); and the cost (in manpower) of using

Technical						Managerial				Human	
Pre-cision	Recall	Cov. Ratio	Time Compl.	Logical Compl.	Auto-mation	Inv. Cost	Oper. Cost	Perva-siveness	State of Develop.	Diff. of Use	Trans-parency
VH	H	H	VH	VH	M	H	H	L	L	M	M

Figure 9. Summary of assessment: denotational semantic methods.

the software library (for storage and retrieval operations). While the first factor is small, the second factor can be fairly large, especially that it depends on an unproven technology. We rate the operating cost of denotational methods as *High*.

3. Pervasiveness. Most of the methods discussed in this section are primarily laboratory prototype – if that; to the best of our knowledge, none are in industrial use of any scale. We rate the pervasiveness of denotational methods as *Low*.

4. State of development. Some signature matching methods [Zaremski and Wing 1995a] qualify as *experimental products*; some functional matching methods [Mili *et al.* 1994] qualify as *laboratory prototypes*; some methods [Massonet and Lamsweerde 1997] are at the stage of research ideas. Overall, we rate the state of development of denotational methods as *Low*.

6.3.3. Human criteria

1. Difficulty of use. Most of the existing prototypes discussed in this section qualify as *easy to use*, given that the reuser is familiar with the appropriate representation of the query (signature, formal functional specification, partially elaborated specification, etc.). Overall, we rate the difficulty of use of denotational methods as *Medium*.

2. Transparency. While little cognizance is required on how the methods operate, much is required on what the results mean and how to interpret them. For example, in order to use the approximate retrieval algorithms of Mili *et al.* [Jilani *et al.* 1997a,b], it is necessary to know what the measures of functional distance represent. We rate the transparency of denotational methods as *Medium*.

The assessment of denotational methods is summarized in figure 9.

7. Topological methods

Topological methods are based on the simple premise that, given a query that describes some required features, we are interested in identifying library assets that come closest to providing these features. Such methods are critically dependent on what it means to *come closest*, which in turn depends on some definition of distance between the query and candidate assets. Topological methods can be divided into two broad categories, according to the precise definition of their retrieval goal:

- *Exclusive approximate retrieval.* Methods that fall into this category make a distinction between two retrieval goals: *exact retrieval*, whereby we seek to identify

library assets that completely satisfy all the requirements of the query; *approximate retrieval*, whereby we seek to identify library components that minimize some measure of distance to the query, once we have established that no asset satisfies it completely. Methods that fall in this category include those of Penix and Alexander [1995, 1996] and Mili *et al.* [Mili *et al.* 1994; Jilani *et al.* 1997a,b].

- *Inclusive approximate retrieval*. Methods that fall into this category make no distinction between exact retrieval and approximate retrieval. Rather, they focus on identifying library assets that minimize some measure of distance to the query, and expect that the outcome will either be an exact match or (failing an exact match) one or more approximate matches. Methods that fall in this category include those of Faustle *et al.* [1996], Girardi and Ibrahim [Girardi and Ibrahim 1994a,b; Girardi 1995] and Spanoudakis and Constantopoulos [1994].

Measures of distance can be divided into two broad classes:

- Measures of *functional* (semantic) distance, which reflect the extent of similarity between the functional properties of the query and those of candidate components.
- Measures of *structural* (syntactic) distance, which reflect the extent of similarity between the structure of (solutions to) the query and the structure of candidate components.

In other words, we can distinguish between these families by the statement that measures of functional distance reflect to what extent the query and candidate assets *act* alike whereas measures of structural distance reflect to what extent the query and candidate assets *look* alike. The assumption that underlies all topological methods is that the closer a candidate component is to a given query, the less effort it takes to modify the component to satisfy the query: it is easy to see why this assumption is more reasonable if we define *closeness* by a structural measure of distance than by a functional measure of distance.

7.1. Literature survey

In [Ostertag *et al.* 1992], Ostertag *et al.* present an AI based library system called AIRS (AI based Reuse System). The AIRS system is based on a hybrid approach, including the faceted index approach [Prieto-Diaz 1990] and the semantic network approach [Devanbu *et al.* 1991; Wood and Sommerville 1988]. The AIRS library contains components and *packages*. The retrieval method is based on the computation of similarity metrics which allow to compare either components or packages. The first metric, used to compare components, is based on the *subsumption* and *closeness* relations. Subsumption applies to the case where a component can be built as a composition of several other components. If the functionality of a component A is partially provided by component B, then A is called the *subsumed* and B the *subsumer*. The second metric, called the *package distance* measures the effort required to implement a target package given a candidate package. Ostertag *et al.* present a

prototype of the AIRS system using the EVB GRACE and the CTC CCIS libraries. The first library includes Ada packages that implement data structures; the second contains C modules that implement basic functionalities of command, control and information systems. The authors describe the classification models used for both libraries.

In [Spanoudakis and Constantopoulos 1993, 1994], Spanoudakis and Constantopoulos introduce a conceptual modeling language (under the name TELOS) which is well adapted to the representation of software artifacts, specifically within the context of object-oriented analysis and design. To this effect, the language supports the representation of object classes, and distinguishes between attributes and entities. Spanoudakis and Constantopoulos use this language to represent queries and assets, and define a measure of structural distance between queries and assets on the basis of an analysis of their TELOS representations. The distance they introduce is a weighted linear combination of four functions which reflect whether relevant entities in the query and the asset are identical and to what extent the query and the asset have common attributes via their shared subclasses and their shared super-classes. Spanoudakis and Constantopoulos built a prototype of their software library on the basis of the foregoing definitions of distance, and integrated it with the existing *Semantic Index System*.

In [Girardi and Ibrahim 1994a,b; Girardi 1995], Girardi and Ibrahim introduce a structural measure of distance between software assets, and use it to perform retrieval in a software library. Library assets are represented using case-frame-like representations that are derived from a declarative definition of the asset in natural language; and queries are derived in a similar fashion from an imperative definition of the desired requirements. The distance between a query and a candidate asset is defined by a linear combination of weighted terms, where each term corresponds to a slot of the case-frame. The term associated to a given slot is the product of two factors: a weight, which reflects the relative importance of the slot in defining the function of the asset; and a similarity index, which reflects to what extent the slot of the query and the slot of the candidate asset are similar. The weight can be determined by the domain analyst who stores the asset in the library, while the similarity index can be retrieved from specialized natural language thesauri. Using existing natural language technology, Girardi and Ibrahim produce a large scale prototype that supports their retrieval method; they call it ROSA (Reuse Of Software Artifacts). Experimentation on 20 queries submitted to a library of 418 general purpose Unix commands produces a recall average of 0.99 and a precision average of 0.89.

In [Penix and Alexander 1995, 1996], Penix and Alexander discuss a formal specification-based method for the storage and retrieval of software components in a software library. In order to improve the recall of their method, they make provisions for approximate retrieval whenever exact retrieval fails to produce assets, or produces too few. Their approximate retrieval operates in a similar manner to their exact retrieval (by matching query features against asset features), but with the difference that query features are generalized, so that the range of assets that satisfy them is widened. Generalization is achieved by weakening the criteria of feature identity and by deleting features from the query. The proximity of a candidate asset to the query is then assessed

by considering the number of query features that are satisfied by the asset, as well as the extent to which they are satisfied; in this sense, the approximate retrieval of Penix and Alexander can be viewed as a topological method.

In [Faustle *et al.* 1996], Faustle *et al.* propose a classification and retrieval method for object-oriented repositories, based on the use of fuzzy logic. The approach is developed within the *Ithaca* application development environment [Ithaca 1993]. The library, called *Software Information Base* (SIB), is organized according to the *Telos* knowledge representation language [Koubarakis 1989]; SIB entries are Telos classes. Each class includes two kinds of knowledge: *functional knowledge* and *teleological knowledge*. Teleological knowledge relates a component with other entities [Constantopolous *et al.* 1993]. A software description consists of a set of keyword pairs also called *features*, which describe the behavior of the asset. The number of features is unbound. Queries have the same structure as assets; they are represented by class attributes and software descriptions. Keywords pairs in a software description are weighted with fuzzy values which correspond to the degree of relevance of the keyword pairs for the user. Similarity between a query and a candidate asset is assessed in terms of a *Confidence Value* (CV), which represents a measure of distance. Faustle *et al.* present a prototype that is used to perform an evaluation of their approach. The experiment presented is based on 87 C++, Smalltalk and Eiffel assets.

In [Jilani *et al.* 1997b], Jilani *et al.* present four measures of semantic distance which reflect four different interpretations of what it means for a component to approximate a query. Components and queries are represented by relational specifications, and the measures of distance take their values in a partially ordered set (rather than the set of real numbers). The measures of distance that are used include: *functional consensus*, which reflects to what extent the query and the asset have common functional features; *refinement difference*, which reflects the amount of functional features of the query that are not satisfied by the candidate asset; *refinement distance*, which reflects the information of the query that is not covered by the asset as well as the information of the asset that is irrelevant to the query; finally, *refinement ratio*, which reflects the functional information that the query and the asset have in common as well as the functional information that sets them apart. Labed *et al.* attempt to automate the process of computing distance and identifying minimal/maximal library components; they do so by means of a theorem prover for first order predicate logic, namely Otter (©Argonne National Laboratory) [McCune 1994]. Experimental results on a small sample library produce interesting results – especially with respect to the interpretation of the measures of distance. In [Jilani *et al.* 1997a], Jilani *et al.* build on their earlier work by attempting to exhibit statistical relationships between the measures of functional distance and the estimated modification effort; they find that some measures of functional distance perform consistently better than others in predicting modification effort.

7.2. Characterizing topological methods

- *Nature of the asset*. Generally, the methods that are based on structural distance make no restriction on the nature of the asset, as they deal with a textual representation of the asset – so that any object that can be represented by a text of the given format qualifies as a possible asset. The methods that are based on functional distance are typically geared towards a specific form of assets, generally executable code – although they can be generalized to accommodate other forms of assets.
- *Scope of the library*. Topological methods are dependent on measures of (structural or functional) distance, which are in turn dependent on specific methods for asset and query representations; indeed it is difficult to imagine a definition of distance that can be meaningfully applied across a wide range of notations. This precludes the application of topological methods across a wide scope of development sites; most typically, these are applicable within a project.
- *Query representation*. Topological methods are not uniform with respect to this characteristic: query representation ranges from TELOS specifications [Faustle *et al.* 1996; Spanoudakis and Constantopoulos 1993, 1994] to case frames [Girardi and Ibrahim 1994a,b] to aggregates of formally specified functional features [Penix and Alexander 1995, 1996] to relational specifications [Jilani *et al.* 1997a,b].
- *Asset representation*. In topological methods, assets and queries have the same representation, and feature the same variance of representation languages; this appears to be a corollary of the need to compute distances (most typically, a distance does not discriminate between its two arguments).
- *Storage structure*. The operation of topological methods is not dependent on storage structure, and with the exception of the method of Faustle *et al.* [1996], there is little mention of any structure of the library in the methods we have discussed above (there *is* discussion of *asset* structure, but not of *library* structure). The only instance where the library does have a meaningful structure is the work of Jilani *et al.* [1997a,b] – even then, the library structure is partially meaningful: the structure can be used to simplify the minimization of functional consensus, but cannot be exploited for the other distances.
- *Navigation scheme*. Because of the absence of structure, the most typical navigation scheme for topological methods is a linear scan of assets in an arbitrary order.
- *Retrieval goal*. Most topological methods are used with the goal of *inclusive approximate retrieval*, whereby, given a query Q, we attempt to identify all the library assets that minimize some measure of distance to Q – with the assumption (hope) that if some assets satisfy the query, then they will be among the assets that are retrieved. Penix and Alexander [1995, 1996] and Jilani *et al.* [1997a,b] use the goal of *exclusive approximate retrieval*, whereby they attempt to find assets that approximate the query whenever it is determined that no asset satisfies the query precisely; also, both Penix *et al.* and Jilani *et al.* have different procedures for exact retrieval and approximate retrieval (rather than to have a single procedure for both).

Attributes	Characterization
Nature of asset	Generally unrestricted.
Scope of library	Within a project.
Query representation	Wide variance, from logic to AI.
Asset representation	Wide variance, from logic to AI.
Storage structure	Typically flat.
Navigation scheme	Linear scan.
Retrieval goal	Inclusive or exclusive approximate retrieval.
Relevance criterion	Minimality of distance.
Matching criterion	The same as relevance criterion.

Figure 10. Characterizing topological methods.

- *Relevance criterion.* In topological methods, one cannot, by observing a query and an asset, determine whether the asset is relevant to the query. Indeed, by their very definition, topological methods identify assets that minimize some measure of distance; hence in order to determine the relevance of a given asset with respect to a query, one must consider (potentially) all the assets of the library. With this qualification in mind, the relevance of an asset C with respect to a query Q is the property that the distance $\delta(Q, C)$ be less than or equal to the quantity $\delta(Q, c)$ for all assets c in the library.
- *Matching criterion.* Topological methods typically implement the relevance criterion verbatim, without computational shortcuts; in fact fairly often the relevance criterion is itself sufficiently simple to make the shortcut futile.

This characterization of topological methods is summarized in figure 10.

7.3. Assessing topological methods

7.3.1. Technical criteria

1. Precision. Because the retrieval goal of topological methods is relatively vague, it is difficult to define – let alone estimate – their precision and recall. An asset C of the library may well minimize some measure of distance to a query Q, and yet be totally irrelevant to the query; this happens, e.g., if the library is (yet) too small, or if it contains nothing that resembles the query. It would be unreasonable to consider such an asset as *relevant* (and count it as a relevant component that has been retrieved); it is equally unreasonable not to count it while it does, in principle, satisfy the relevance criterion. To account for our inability to make a judgment on the precision of this family of methods, we rate this precision as *Unknown* – notwithstanding that Girardi and Ibrahim report experimental results that would rate their method's precision as *Very High*.

2. Recall. Likewise, for the same reasons as above, we rate the recall of topological methods as *Unknown* – notwithstanding that Girardi and Ibrahim report experimental results that would rate their method's recall as *Very High*.

3. Coverage ratio. In view of the navigation scheme of topological methods, we rate their coverage ratio as *Very High*.

4. Time complexity per match. The time complexity of a single distance evaluation in topological methods varies from a linear function of the size of the query [Faustle *et al.* 1996; Girardi and Ibrahim 1994a,b; Penix and Alexander 1995, 1996; Spanoudakis and Constantopoulos 1993, 1994] to an exponential functional of the size of the query [Jilani *et al.* 1997a,b] driven by the combinatorial explosion of the theorem proving process. Because a match involves typically distance comparison with many other retrieval candidates, potentially all of the assets in the library, this adds a factor of N to the time complexity of distance evaluation (where N is the size of the library). We rate the time complexity of topological methods as *High*. We reserve the rating *Very High* to denotational methods of approximate retrieval, which combine the obligation to compare distances across the whole library with the time complexity of individual distance evaluations.

5. Logical complexity per match. The logical complexity per match of topological methods varies from compound boolean expressions used to compute and compare numeric distances [Faustle *et al.* 1996; Girardi and Ibrahim 1994a,b; Spanoudakis and Constantopoulos 1993, 1994] to compound predicate logic expressions [Jilani *et al.* 1997a,b; Penix and Alexander 1995, 1996] used to check correctness-like properties and evaluate relational measures of distance. On balance, we rate the logical complexity of topological methods as *Medium*.

6. Automation/automation potential. Virtually all of the methods discussed in this section have resulted in an automated tool, sometimes an industrial-strength tool [Girardi and Ibrahim 1994a,b]. We rate the automation of topological methods as *High*.

7.3.2. Managerial criteria

1. Investment cost. Virtually all topological methods involve a nontrivial investment cost, in terms of setting up the software library and the operational procedures and organizational provisions that are required to operate it. In particular, all methods involve a specific notation for specifying and representing assets, as well as a nontrivial amount of tool acquisition and personnel training. We rate the investment cost of topological methods as *Very High*.

2. Operating cost. If we take into account the overhead required to maintain a software library that implements a topological method, as well as the cost of performing a retrieval from such a library, we realize that it is fairly costly to operate a topological library. Indeed inserting an asset in a library involves a non-trivial amount of work (representing the asset in the proposed notation). On the other hand, a retrieval from a topological library is potentially expensive (scanning all components, computing their distance to the query, comparing pairwise distances to the query, sometimes using a

Technical						Managerial				Human	
Pre-cision	Recall	Cov. Ratio	Time Compl.	Logical Compl.	Auto-mation	Inv. Cost	Oper. Cost	Perva-siveness	State of Develop.	Diff. of Use	Trans-parency
U	U	VH	H	M	H	VH	VH	L	L	VH	VH

Figure 11. Summary of assessment: topological methods.

theorem prover). Hence we rate the operating cost of topological libraries as *Very High*.

3. Pervasiveness. To the best of our knowledge, and with the possible exception of the libraries of Spanoudakis and Constantopoulos [1993, 1994] and Faustle *et al.* [1996] (which have been developed as part of multi-national European projects), most topological libraries have been used only within their native laboratory. Hence we rate the pervasiveness of topological methods as *Low*.

4. State of development. To the best of our knowledge, and with the possible exception of the library of Girardi and Ibrahim [1994a,b] (which relies on large scale thesauri), most topological methods are mere laboratory prototypes. Hence we rate the state of development of topological methods as *Low*.

7.3.3. Human criteria

1. Difficulty of use. Two sources of difficulty arise when using topological methods: the first stems from the specific notations that must be used to represent queries; the second stems from the need to have a detailed understanding of the measure of distance that is used in the minimization process. The result of the retrieval is meaningful only to the extent that the user can interpret it properly; this, in turn, is only possible if the user understands the meaning of the measure of distance that is used, and knows how to interpret it in practice. Hence we rate the difficulty of use of topological methods as *Very High*.

2. Transparency. The user of a topological method is not required to have any cognizance of the operation of the retrieval procedure. Hence we rate the transparency of topological methods as *Very High*.

This assessment of topological methods is summarized in figure 11.

8. Structural methods

It is fair to say that all software library organizations we have discussed so far select candidate library components solely on the basis of their function: we decide whether to select a component by matching the functional properties of the candidate component against desired functional features. An alternative rationale is to select candidates not on the basis of their function but rather on the basis of their structure: we select a component whenever we have reason to believe that a possible solution

to our query has the same structure as the component under consideration. If our intent is to use retrieved components `verbatim` (i.e., without modification), then the functional criterion is best suited; if, however, our intent is to use retrieved components after modification, then it is best to seek components whose structure looks as close as possible to the target programming solution, so as to minimize the modification effort. It is not uncommon to use the functional criterion even when we intend to use components after modification: the rationale of this approach is the assumption that whenever two components have similar functional properties, they also have similar structures; while this assumption is not always borne out in practice, it is not totally unreasonable.

To synthesize our discussion, there are two distinct families of criteria that we can use to select candidate components in a software library: *functional criteria*, which seek to identify components that have the same functional properties as the query (*act like* the query); *structural criteria*, which seek to identify components that have the same structure as possible solutions to the query (*look like* the query). Typically functional criteria are best adapted to black box reuse, whereby components are used verbatim, and structural criteria are best adapted to white box reuse, whereby components are used after modification – although it is not uncommon that functional criteria are used for both paradigms of reuse. In order to illustrate the contrast between these two families of criteria, we consider in turn two examples:

- We consider two distinct sorting programs, e.g., a quicksort and a selection sort: even though they have the same function (act alike), they have very distinct structures (do not look alike).
- We consider two programs S and P, where S computes the sum of an array and P computes the product of an array: typically these programs have the same structure (look alike), yet they have different functions (do not act alike: consider that, except for signature matching, none of the retrieval algorithms we have discussed so far would, or should, retrieve P for query S).

In this section, we discuss a set of software library organizations that attempt to retrieve software components on the basis of their structure.

8.1. Literature survey

In [Rich and Schrobe 1976, 1978; Rich and Waters 1990; Waters 1981, 1982], Rich *et al.* discuss a long-term research project titled *The Programmer's Apprentice*. The project stems from the observation that high level programming languages have produced great gains in program quality and programmer productivity by delegating unimportant implementation decisions to the compiler, and that equally dramatic gains are possible if we could delegate more clerical steps to automated programming assistants, such as the Programmer's Apprentice. For the purpose of our survey, this project is interesting because it includes a library of *programming clichés*. The clichés are generic algorithmic or data patterns; they provide the basic vocabulary used in

the communication between the programmer and the Programmer's Apprentice, and embodies most of their shared knowledge. In [Waters 1981] a variation on Ada's procedure notation is used to represent clichés. This form specifies the name of the cliché, some declarations that define the important agents of the cliché, as well as the computation that correspond to the cliché. Two features of the Ada-like representation of clichés provide genericity: *roles*, which are similar to Ada's formal parameters but are substantially more flexible; and *overloading*, which allows general purpose functions to be defined in the absence of complete type information. Clichés are stored in a library that is structured by the hierarchical genericity relation, and are retrieved by matching their name against queries that are submitted in pseudo-natural language. Examples of clichés are: *EqualityWithinEpsilon*, which checks that two arguments (that instantiate roles) are within some ϵ value of each other; *FileEnumeration*, which sequentially enumerates all the records of a file; *SimpleReport*, which produces a report from a file, according to a predefined format. We consider that this library organization is *structure*-based because structure is the most important feature of a cliché: by instantiating the roles of a cliché, we get a wide range of different functions; what the instantiated clichés share is a common structure. Also, it is this structure that the reuser refers to when he/she invokes the name of the cliché.

Building on the results achieved with the *Programmer's Apprentice*, Rich and Waters [1988] discuss research ideas towards the *Design Apprentice*. Their aim is to have the Design Apprentice embody the following capabilities:

- The ability to analyze and understand specifications written in a declarative language.
- The ability to detect errors made by the programmer, and to produce explanations thereof.
- The ability to automatically select reasonably efficient implementations.

The knowledge of the Design Apprentice is embodied in its clichés for typical specifications, typical designs, and typical hardware characteristics. Examples of specification clichés in the device driver application domain include: *initializing*, *reading*, *writing*, *opening*, and *closing* a device. Examples of design clichés include the generic *device driver* and its specializations *printer driver* and *interactive display driver*. Examples of hardware clichés include *serial line unit*, *printer* and *interactive display device*. As it is envisioned in [Rich and Waters 1988], the Design Apprentice analyzes queries submitted by the software designer and selects design clichés from a library, which is structured by the *IsA* relation. The selected clichés are instantiated and combined on the basis of an interactive dialogue with the designer. Recent instances of design pattern libraries include the works of Gomaa and Farrukh [1997] (with an emphasis on distributed applications), Flener *et al.* [1997] (with an emphasis on logic programming design patterns), and Eden *et al.* [1997] (with an emphasis on object-oriented design patterns).

In [Reubenstein and Waters 1991], Reubenstein and Waters present the *Requirements Apprentice*, an automated assistant for requirements acquisition. Reubenstein and Waters identify three phases in the requirements acquisition process: *elicitation*, *formalization* and *validation*. They focus their attention on the second phase, which bridges the gap between an informal and a formal specification. As with the previous *Apprentices*, the Requirements Apprentice centers on the codification of relevant knowledge by means of clichés. Requirements clichés are stored in a library, which is organized as three separate hierarchically structured sections (based on the theory that requirements are composed of three kinds of information): the *environment* section describes general knowledge about the application domain; the *needs* section gives a high-level description of the desires of the end-users; the *system* section contains a high-level specification for a system that meets the needs. Clichés have the same overall structure as with the earlier *Apprentices*, and are represented by the AI construct of *frame*, whereby cliché roles are represented by slots. Requirement clichés are identified by their names and are retrieved by parsing natural language-like user queries.

In [Paul and Prakash 1994], Paul and Prakash discuss the retrieval of source code from a software library using structural information. Paul and Prakash identify the following goals for structure-based component retrieval in a software library: *Reengineering code*, *Making queries on programs*, and *Understanding programs*. They define a sophisticated language-dependent notation to represent structural patterns of source code, and define a pattern matching mechanism that determines whether a piece of source code matches a given pattern. The notation allows arbitrarily nondeterministic specifications of the desired pattern, and the matching mechanism makes provisions for variable renaming and statement equivalence. To illustrate their approach, Paul and Prakash implement it for two programming languages, including C, and run experiments of structural matching; because the matching is primarily a syntactic analysis task, it can be done very efficiently with today's parsing technology.

Nelson and Poulis [1995] present a method specifically geared towards object-oriented programming. In their Computer Aided Prototyping System (CAPS) components (Ada code) are described in the high level specification language PSDL. Since PSDL contains basically all the information contained in the interface definition of a class, their CSRS (Class Storage and Retrieval System) can take full advantage of the structure of class lattices.

8.2. Characterizing structural methods

We review in turn the main attributes of software libraries, as we introduced them in section 2.3, and discuss how structural methods can be characterized in terms of these attributes.

- *Nature of the asset*. Structural methods can accommodate a wide range of assets, including source code [Paul and Prakash 1994], programming clichés [Rich and

Schrobe 1976, 1978; Waters 1981, 1982; Rich and Waters 1990], design clichés [Rich and Waters 1988] and requirements clichés [Reubenstein and Waters 1991].

- *Scope of the library*. The *Apprentice* libraries [Reubenstein and Waters 1991; Rich and Schrobe 1976, 1978; Rich and Waters 1988, 1990; Waters 1981, 1982] are supposed to hold general programming knowledge, hence have (by design) a wide scope. However they can probably be made more efficient if they were application domain-specific, especially because they depend for their operation on verbal communication with the reuser, hence work best with a limited vocabulary. In principle, the solution of Paul and Prakash [1994] is only limited by the programming language; for a given programming language (e.g., C), it can have an arbitrarily large scope.
- *Query representation*. In the *Apprentice* libraries, queries are cliché names, which are supposed to be shared between the system and the reuser. In Paul and Prakash's library, queries are syntactic patterns formulated in a special purpose language.
- *Asset representation*. The assets of Paul and Prakash's library are represented in source language code. The assets in the *Apprentice* libraries are represented by clichés: a cliché is composed of *roles* and *constraints*; the roles of a cliché are the parts that vary from one instance of the cliché to the next, and the constraints specify how the roles interact and place limits on the parts that can fill the roles. Clichés are given names, and it is assumed that the user of a cliché library knows clichés by their names.
- *Storage structure*. Storage structure is not an important feature of structural methods – asset structure is. Some cliché libraries are structured according to the *IsA* relation, but this relation is usually fairly shallow (does not contain long paths).
- *Navigation scheme*. Structural methods typically scan the library entries in a predefined but non-relevant order.
- *Retrieval goal*. In general terms, the goal of structural retrieval is to identify assets that have a predefined structure; the assumption is that we will subsequently customize the selected asset(s) by filling its *roles*.
- *Relevance criterion*. Given a query Q, an asset A is considered relevant with respect to Q if and only if A has the structure dictated by Q.
- *Matching criterion*. The *Apprentice* methods consider that the association between the name of a cliché and its structure is part of the common knowledge that is shared by the *Apprentice* and the reuser, hence retrieve clichés by their name (in order for a query Q to match an asset A, Q has to refer to the name of A). The method of Paul and Prakash matches a query against an asset by parsing the asset according to the prescriptions of the query.

Figure 12 summarizes our discussions in tabular form.

Attributes	Characterization
Nature of asset	Source code, or clichés.
Scope of library	Typically unlimited.
Query representation	Name of a cliché or syntactic pattern.
Asset representation	Source code, or roles and constraints.
Storage structure	Typically flat. Some *IsA* relations.
Navigation scheme	Typically exhaustive.
Retrieval goal	Retrieving all components of a given structure.
Relevance criterion	Structural match.
Matching criterion	Name identity or successful parse.

Figure 12. Characterizing structural methods.

8.3. Assessing structural methods

8.3.1. Technical criteria

1. Precision. The precision of the *Apprentice* libraries is totally dependent on the reuser's ability to remember the names of the clichés that he needs for a particular application. As far as the library is concerned, the precision is 1. Because it relies on a highly reliable technology (syntactic analysis), the parsing technique of Paul and Prakash also has a precision of 1, giving a rating of *Very High*.

2. Recall. For the same reasons that we discussed above (for precision), the matching condition of structural methods does not miss relevant components. Given that, in addition, the navigation scheme provides for visiting virtually all the entries of the library, the recall of structural methods is at or near 1. We rate the recall of structural methods as *Very High*.

3. Coverage ratio. Except for skipping some portions of the (shallow) *IsA* hierarchy in the *Apprentice* libraries, structural methods have a coverage ratio of 1. We rate the coverage ratio of structural methods as *Very High*.

4. Time complexity per match. In the *Apprentice* libraries the match is a trivial comparison of names – which follows a non-trivial but simple parse of a natural language-like query. The parsing technique of Paul and Prakash is at worst linear as a function of the size of the structural pattern submitted as a query. Overall we rate the time complexity of structural methods as *Very Low*.

5. Logical complexity per match. In all structural methods, the match involves some syntactic analysis: In the *Apprentice* libraries, the query is analyzed to identify the names of library clichés, and in the method of Paul and Prakash the query provides a syntactic template against which candidate components are matched. We rate the logical complexity of structural methods at *Low*.

6. Automation/automation potential. Syntactic analysis is one of the most straightforward areas of software production; it has been investigated thoroughly and automated

tools for this task are widely available. We rate the automation potential of structural methods at *Very High*.

With high precision, recall, and automation potential, and low time complexity and logical complexity, structural methods look like ideal methods – but they are not necessarily; judgment on this matter must be reserved until we investigate other criteria.

8.3.2. Managerial criteria

We consider how structural methods fare with respect to managerial criteria.

1. Investment cost. A good deal of domain analysis is required before one can set up a cliché library: domain analysis is required to define abstract, generic clichés, to define appropriate roles for them, and to ensure their orthogonality and their coverage. We rate structural methods as *Medium* with respect to investment costs.

2. Operating cost. By design, clichés are generic, abstract units that can be instantiated into a wide range of usable assets. By virtue of instantiation, a wide range of assets can be supported by a small number of clichés, hence producing *Low* operating costs.

3. Pervasiveness. To the best of our knowledge, all the methods discussed in this section are not widely used outside their birthplace. We rate these methods as *Low* with respect to pervasiveness.

4. State of development. Likewise, we know of no product that supports structural methods beyond the laboratory prototypes that are discussed in the research literature; hence we rate these methods as *Low* with respect to state of development.

8.3.3. Human criteria

We consider how structural methods fare with respect to managerial criteria.

1. Difficulty of use. The *Apprentice* methods interact with the user by means of a natural language-like notation, hence are fairly easy to use; the syntactic method of Paul and Prakash is also easy to use, once one gets past the simple notation of syntactic patterns. We rate structural methods as *Very Low* with respect to difficulty of use.

2. Transparency. Transparency is perhaps the criterion where structural methods fare worst: One cannot use *Apprentice* methods unless one has a detailed knowledge of all the relevant clichés of the library, including their structure, their roles, their constraints, and their name; likewise, one cannot use Paul and Prakash's syntactic method unless one has already covered a lot of ground in the design of the product at hand, and has decided on the structure of the component to be retrieved. We rate the transparency of structural methods as *Very Low*.

Figure 13 summarizes our assessment of the structural methods.

Technical						Managerial				Human	
Pre-cision	Recall	Cov. Ratio	Time Compl.	Logical Compl.	Auto-mation	Inv. Cost	Oper. Cost	Perva-siveness	State of Develop.	Diff. of Use	Trans-parency
VH	VH	VH	VL	L	VH	M	L	L	L	VL	VL

Figure 13. Summary of assessment: structural methods.

9. Conclusion

9.1. Summary of contributions

Despite several years of active research, the storage and retrieval of software assets in general and programs in particular remains an open problem. While there is a wide range of solutions to this problem, many of which have led to operational systems, no solution offers the right combination of efficiency, accuracy, user-friendliness and generality to afford us a breakthrough in the practice of software reuse. Generally speaking, most solutions are either too inaccurate to be useful or too intractable to be usable. Given the amount of research effort invested in this area, this could be viewed as a meager balance sheet.

In order to orient future research in this area, we have, in this paper, presented a survey of existing solutions. More than the survey itself, we want to think that the main contributions of our paper is in:

- Characterizing software libraries by means of (fairly) orthogonal features; these allow researchers to better define their solutions and classify them.
- Using this characterization to classify existing software libraries into six broad families. The definition of the six families is controversial, and so is the assignment of methods to families – but it is a working solution.
- Defining criteria by which software libraries can be evaluated; the traditional criteria of precision and recall are important, but they are not sufficient. We have defined a set of twelve criteria, that we divided into technical, managerial, and human criteria.

Traditionally, papers that discuss software libraries have a related work section where they discuss a few methods and compare them to their own. With this survey, we aim to streamline such comparisons by allowing authors to better characterize their method, perhaps identify the family that their method belongs to, then compare their methods to other members of the same family.

9.2. Summary of evaluations

Figure 14 summarizes our assessment of all the families of software libraries that we have discussed in this paper. We have resisted the temptation to convert our ratings into numeric values and to rank families of methods by their grade point averages, first because assessing software libraries is really a multi-criteria process (twelve criteria, in this case), and we wish to keep it that way; second because these criteria are

	Technical						Managerial				Human	
Method	Pre-cis.	Re-call	Cov. Ratio	Time Compl.	Log. Compl.	Auto-mat.	Inv. Cost	Oper. Cost	Perva-siv.	St. of Develop.	Diff. Use	Trans-par.
Inf. Ret.	M	H	L	L	M	H	VL	L	H	H	M	H
Descr.	H	H	VH	VL	L	VH	H	H	H	H	VL	VH
Operat.	VH	H	H	M	M	VH	L	M	M	M	L	VH
Denot.	VH	H	H	VH	VH	M	H	H	L	L	M	M
Topol.	U	U	VH	H	M	H	VH	VH	L	L	VH	VH
Struct.	VH	VH	VH	VL	L	VH	M	L	L	L	VL	VL

Figure 14. Summary of assessment: all methods.

neither perfectly orthogonal nor equally weighted; third, because specific methods or implementations may perform significantly better or significantly worse than the rating we have assigned to the class of the method as a whole. A summary analysis does show some edge of simpler methods, but this must be qualified extensively, and should not be viewed as discouraging the investigation of sophisticated methods.

9.3. Surveys of software libraries

In [Frakes and Pole 1990], Frakes and Pole survey some methods of software storage and retrieval, using a classification into three families: *library and information science indexing methods*, *Knowledge-based methods*, and *Hypertext methods*. In [Frakes and Pole 1994], Frakes and Pole revisit the classification and propose a set of assessment criteria that include, in addition to precision and recall: *effectiveness*, *overlaps* and *searching time*.

Krueger [1992] and Mili *et al.* [1995] present surveys of software reuse in general, in which they discuss the storage and retrieval of software assets for the purpose of reuse. Because software libraries are not the focus of their surveys, they are treated at a general level, and account for the sate of the art at their time of publication.

In [Atkinson 1996], Atkinson defines an abstract retrieval framework using the specification language Z, and attempts to model a number of existing storage and retrieval methods as specialized instances of the abstract framework. Among the methods that he models are: specification matching (an instance of denotational methods, discussed in section 6), behavioral retrieval (an instance of operational methods, discussed in section 5), and faceted retrieval (an instance of descriptive methods, discussed in section 4).

References

Allemang, D.T. (1990), "Understanding Programs as Devices," PhD Thesis, Ohio State University, Columbus, OH.

Allemang, D.T. (1991), "Use of Functional Models in the Domain of Automatic Debugging," *IEEE Expert 6*, 6, 13–18.

ASSET (1993), "ASSET Submittal Guidelines," Technical Report SAIC-92/7625-00, version 1.2, SAIC/ASSET, Morgantown, WV, December 1993.

Atkinson, S. (1996), "A Formal Model for Integrated Retrieval from Software Libraries," In *Technology of Object-Oriented Languages and Systems: TOOLS 21*, Prentice-Hall, Englewood Cliffs, NJ.

Atkinson, S. and R. Duke (1994), "A Methodology for Behavioral Retrieval from Class Libraries," *Australian Computer Science Communications 17*, 1, 13–20.

Badaro, N. and T. Moineau (1991), "ROSE-Ada: A Method and a Tool to Help Reuse Ada Code," In *Proceedings of the Ada: The Choice for '92*, LNCS 499, Springer, Berlin, Germany, 1991.

Boehm, B.W. (1981), *Software Engineering Economics*, Prentice-Hall, Englewood Cliffs, NJ.

Boerstler, J. (1995), "Feature Oriented Classification for Software Reuse," In *Proceedings of the 7th International Conference on Software Engineering and Knowledge Engineering*, SEKE '95, KSI Knowledge Systems Institute, Skokie, IL, pp. 204–211.

Booch, G. (1994), *Object-Oriented Analysis and Design with Applications*, Benjamin/Cummings Publishing Company, Redwood City, CA.

Boudriga, N., A. Mili and R.T. Mittermeir (1992), "Semantic-Based Software Retrieval to Support Rapid Prototyping," *Structured Programming 13*, 109–127.

Boudriga, N., A. Mili and R. Zalila (1992a), "An Automated Tool for Specification Validation: Design and Preliminary Implementation," In *Proceedings of the 25th Hawaii International Conference on System Sciences*, IEEE Computer Society Press, Los Alamitos, CA, pp. 74–82.

Boudriga, N., A. Mili, R. Zalila and F. Mili (1992b), "A Relational Model for the Specification of Data Types," *Computer Languages 17*, 2, 101–131.

Brooks, F.P. (1987), "No Silver Bullet: Essence and Accidents of Software Engineering," *IEEE Computer Magazine 20*, 4, 10–19.

Browne, S.V. and J.W. Moore (1997), "Reuse Library Interoperability and the World Wide Web," In *Proceedings of the ACM SIGSOFT Symposium on Software Reusability*, SSR '97, *ACM Software Engineering Notes 22*, 3, 182–189.

Castano, S., V. De Antonellis and B. Pernici (1996), "Criteria and Metrics for Quantifying Similarity Factors," In *Information Processing and Management of Uncertainty in Knowledge-Based Systems, Proceedings of the 6th IPMU '96*, Volume III, Universidad de Granada, Granada, Spain, pp. 1091–1097.

Celentano, A., M.G. Fugini and S. Pozzi (1995), "Knowledge-Based Document Retrieval in Office Environments: The Kabiria System," *ACM Transactions on Information Systems 13*, 3, 237–268.

Chen, Y. and B. Cheng (1997), "Facilitating an Automated Approach to Architecture-Based Software Reuse," In *Proceedings of the 12th IEEE International Automated Software Engineering Conference*, IEEE Computer Society Press, Los Alamitos, CA, pp. 238–246.

Cheng, B.H.C. and J.J. Jeng (1992a), "Formal Methods Applied to Reuse," In *Proceedings of the 5th Annual Workshop on Software Reuse*, WISR-5, Palo Alto, CA, `ftp://gandalf.umcs.maine.edu/pub/WISR/wisr5/proceedings/`.

Cheng, B.H.C. and J.J. Jeng (1992b), "Reusing Analogous Components," In *Proceedings of the IEEE International Conference on Tools with AI*, IEEE Computer Society Press, Los Alamitos, CA.

Chou, C.S., J.Y. Chen and C.G. Chung (1996), "A Behavior-Based Classification and Retrieval Technique for Object-Oriented Specification Reuse," *Software – Practice and Experience 26*, 7, 815–832.

Clifton, C. and W.-S. Li (1995), "Classifying Software Components Using Desgn Characteristics," In *Proceedings of the 10th Knowledge-Based Software Engineering Conference*, KBSE '95, IEEE Computer Society Press, Los Alamitos, CA, pp. 139–146.

Constantopoulos, P., M. Doerr and Y. Vassiliou (1993), "Repositories for Software Reuse: The Software Information Base," In *Information System Development Process*, N. Prakash, C. Rolland and B. Pernici, Eds., North-Holland, Amsterdam, The Netherlands.

Devanbu, P., R. Brachman, P. Selfridge and B. Ballard (1991), "Lassie: A Knowledge-Based Software Information System," *Communications of the ACM 34*, 5, 34–39.

Eden, A., A. Yehudai and J. Gil (1997), "Precise Specification and Automatic Application of Design Patterns," In *Proceedings of the 12th IEEE International Automated Software Engineering Conference*,

IEEE Computer Society Press, Los Alamitos, CA, pp. 143–152.
Faison, T. (1997), "Interactive Component-Based Software Development with Expresso," In *Proceedings of the 12th IEEE International Automated Software Engineering Conference*, IEEE Computer Society Press, Los Alamitos, CA, pp. 293–294.
Faloutsos, C. (1985), "Access Methods for Text," *ACM Computing Surveys 17*, 1, 49–74.
Faustle, S., M.G. Fugini and E. Damiani (1996), "Retrieval of Reusable Components using Functional Similarity," *Software Practice and Experience 26*, 5, 491–530.
Fickas, S. (1997), "Workshop Software on Demand: Issues for Requirements Engineering," In *Proceedings of the 3rd IEEE International Symposium on Requirements Engineering*, RE '97, IEEE Computer Society Press, Los Alamitos, CA, pp. 222–223.
Fischer, B., M. Kievernagel and G. Snelting (1995), "Deduction-Based Software Component Retrieval," In *Proceedings of the IJCAI Workshop on Reuse of Proofs, Plans and Programs*, German Research Center for Artificial Intelligence, Saarbrücken, Germany, pp. 6–10.
Flener, P., K. Lau and M. Ornaghi (1997), "Correct-Schema-Guided Synthesis of Steadfast Programs," In *Proceedings of the 12th IEEE International Automated Software Engineering Conference*, IEEE Computer Society Press, Los Alamitos, CA, pp. 153–160.
Frakes, W.B. (1991), "Software Reuse: Payoff and Transfer," In *Proceedings of the 8th AIAA Computing in Aerospace Conference*, Baltimore, MD, pp. 829–831.
Frakes, W.B. (1992), "Introduction to Information Storage and Retrieval Systems," In *Information Retrieval: Data Structures and Algorithms*, W.B. Frakes and R. Baeza-Yates, Eds., Prentice-Hall, Upper Saddle River, NJ, Chapter 1, pp. 1–12.
Frakes, W.B. and R. Baeza-Yates, Eds. (1992), *Information Retrieval: Data Structures and Algorithms*, Prentice-Hall, Upper Saddle River, NJ.
Frakes, W.B. and C.J. Fox (1993), "Software Reuse Survey Report," Technical Report, Software Engineering Guild, Sterling, VA.
Frakes, W.B. and P.B. Gandel (1990), "Representing Reusable Software," *Information and Software Technology 32*, 10, 653–664.
Frakes, W.B. and B.A. Nejmeh (1987a), "An Information System for Software Reuse," In *Proceedings of the 10th Minnowbrook Workshop on Software Reuse*, Syracuse University, Minnowbrook, NY.
Frakes, W.B. and B.A. Nejmeh (1987b), "Software Reuse Through Information Retrieval," In *Proceedings of the 20th Annual Hawaii International Conference on System Sciences*, IEEE Computer Society Press, Los Alamitos, CA, pp. 530–535.
Frakes, W.B. and T.P. Pole (1990), "Proteus: A Reuse Library Systems that Supports Multiple Representation Methods," *ACM SIGIR Forum 24*, 43–55.
Frakes, W.B. and T.P. Pole (1994), "An Empirical Study of Representation Methods for Reusable Software Components," *IEEE Transactions on Software Engineering 20*, 8, 617–630.
Gaudel, M.C. and Th. Moineau (1988), "A Theory of Software Reusability," In *Proceedings of the 2nd European Symposium on Programming, ESOP '88*, LNCS 300, Springer, Berlin, Germany, pp. 115–130.
Girardi, M.R. (1995), "Classification and Retrieval of Software through Their Description in Natural Language," Technical Report 2782, University of Geneva, Geneva, Switzerland.
Girardi, R. and B. Ibrahim (1994a), "Automatic Indexing of Software Artifacts," In *Proceedings of the 3rd International Conference on Software Reuse*, IEEE Computer Society Press, Los Alamitos, CA, pp. 24–32.
Girardi, R. and B. Ibrahim (1994b), "A Similarity Measure for Retrieving Software Artifacts," In *Proceedings of the 6th International Conference on Software Engineering and Knowledge Engineering*, SEKE '94, Knowledge Systems Institute, Skokie, IL, pp. 89–100.
Gomaa, H. and G. Farrukh (1997), "Automated Configuration of Distributed Applications from Reusable Software Architectures," In *Proceedings of the 12th IEEE International Automated Software Engineering Conference*, IEEE Computer Society Press, Los Alamitos, CA, pp. 193–200.

Gomez, H. (1997), "A Cognitive Approach to Software Reuse Applying Case-Based Reasoning to Mutant Cases," In *Proceedings of the 9th International Conference on Software Engineering and Knowledge Engineering*, SEKE '97, Knowledge Systems Institute, Skokie, IL, pp. 512–519.

Gonzalez, P.A. and C. Fernandez (1997), "A Knowledge-Based Approach to Support Software Reuse in Object-Oriented Libraries," In *Proceedings of the 9th International Conference on Software Engineering and Knowledge Engineering*, SEKE '97, Knowledge Systems Institute, Skokie, IL, pp. 520–527.

Guttag, J., J.J. Horning and J.M. Wing (1985), "The Larch Family of Specification Languages," *IEEE Software 2*, 5, 24–36.

Hall, R.J. (1993), "Generalized Behaviour-Based Retrieval," In *Proceedings of the 15th International Conference on Software Engineering*, IEEE Computer Society Press, Los Alamitos, CA.

Helm, R. and Y.S. Maarek (1991), "Integrating Information Retrieval and Domain Specific Approaches for Browsing and Retrieval in Object-Oriented Class Libraries," In *Proceedings of the OOPSLA '91, SigPlan Notices 26*, 11, 47–61.

Hoare, C.A.R. (1969), "An Axiomatic Basis for Computer Programming," *Communications of the ACM 12*, 10, 576–583.

Hochmüller, E. (1992), "AUGUSTA – Eine Reuse-Orientierte Software-Entwicklungsumgebung zur Erstellung von Ada-Applikationen," PhD Thesis, Universität Klagenfurt, Klagenfurt, Austria.

Ihrig, L. and S. Kambhampati (1995), "Automatic Storage and Indexing of Plan Derivations Based on Replay Failures," In *Proceedings of the IJCAI Workshop on Reuse of Proofs, Plans and Programs*, German Research Center for Artificial Intelligence, Saarbrücken, Germany.

Isakowitz, T. and R.J. Kauffman (1996), "Supporting Search for Reusable Software Objects," *IEEE Transactions on Software Engineering 22*, 6, 407–423.

Ithaca (1993), "Integrated Toolkit for Highly Advanced Computer Applications," Technical Report 2705, EEC-Esprit II Project.

Jeng, J.J. and B.C.H. Cheng (1995), "Specification Matching for Software Reuse: A Foundation," In *Proceedings of the ACM SIGSOFT Symposium on Software Reuse*, SSR '95, ACM Press, New York, NY, pp. 97–105.

Jilani, L.L., R. Mili, M. Frappier, J. Desharnais and A. Mili (1997a), "Retrieving Software Components That Minimize Adaptation Effort," In *Proceedings of the 12th IEEE International Automated Software Engineering Conference*, ASE '97, IEEE Computer Society Press, Los Alamitos, CA, pp. 255–262.

Jilani, L.L., R. Mili and A. Mili (1997b), "Approximate Retrieval: An Academic Exercise or a Practical Concern?" In *Proceedings of the 8th Annual Workshop on Software Reuse (WISR-8)*, `ftp://gandalf.umcs.maine.edu/pub/WISR/wisr8/proceedings/`.

Kaplan, S.M. and Y.S. Maarek (1990), "Incremental Maintenance of Semantic Links in Dynamically Changing Hypertext Systems," *Interacting with Computers 2*, 3.

Karlsson, E.-A. (1995), *Software Reuse – A Holistic Approach*, Wiley, Chichester, West Sussex, UK.

Khoshgoftaar, T.M., K. Ganesan, E.B. Allen, F.D. Ross, R. Munikoti, N. Goel and A. Nandi (1997), "Predicting Fault Prone Modules with Case-Based Reasoning," In *Proceedings of the 8th International Symposium on Software Reliability Engineering*, IEEE Computer Society Press, Los Alamitos, CA.

Kolbe, T. and Ch. Walther (1995), "Proof Management and Retrieval," In *Proceedings of the IJCAI Workshop on Reuse of Proofs, Plans and Programs*, IEEE Computer Society Press, Los Alamitos, CA.

Koubarakis, M. (1989), "TELOS: Features and Formalization," Technical Report KRR-TR-89-1, University of Toronto, Toronto, Ontario, Canada.

Krueger, Ch.W. (1992), "Software Reuse," *ACM Computing Surveys 24*, 2, 131–183.

Liver, B. (1996), "A Functional Model of Software and Its Application in Troubleshooting, Design, and Reuse," PhD Thesis, EPFL Lausanne, Lausanne, Switzerland.

Liver, B. (1997), "Introduction to Reasoning about Functions for Design Patterns and Modules," In *Proceedings of the 9th International Conference on Software Engineering and Knowledge Engineering*, SEKE '97, Knowledge Systems Institute, Skokie, IL, pp. 148–157.

Lucarella, D. and A. Zanzi (1996), "A Visual Retrieval Environment for Hypermedia Information Systems," *ACM Transactions on Information Systems 14*, 1, 3–29.

Lung, C.-H. and J.E. Urban (1995), "An Approach to the Classification of Domain Models in Support of Analogical Reuse," In *Proceedings of the ACM SigSoft Symposium on Software Reusability*, SSR '95, ACM Press, New York, NY, pp. 169–178.

Maarek, Y.S. and D.M. Berry (1989), "The Use of Lexical Affinities in Requirements Extraction," In *Proceedings of the 5th International Workshop on Software Specification and Design*, IWSSD-5, IEEE Computer Society Press, Los Alamitos, CA, pp. 196–202.

Maarek, Y.S., D.M. Berry and G.E. Kaiser (1991), "An Information Retrieval Approach for Automatically Constructing Software Libraries," *IEEE Transactions on Software Engineering 17*, 8, 800–813.

Maiden, N.A.M., P. Mistry and A.G. Sutcliffe (1995), "How People Categorise Requirements for Reuse: A Natural Approach," In *Proceedings of the 2nd IEEE International Symposium on Requirements Engineering*, RE '95, IEEE Computer Society Press, Los Alamitos, CA, pp. 148–155.

Maiden, N.A.M. and A.G. Sutcliffe (1992), "Exploiting Reusable Specifications Through Analogy," *Communications of the ACM 35*, 4, 55–64.

Maiden, N.A.M. and A.G. Sutcliffe (1993), "Requirements Engineering by Example," In *Proceedings of the 1st International Symposium on Requirements Engineering*, RE '93, IEEE Computer Society Press, Los Alamitos, CA, pp. 104–112.

Massonet, P. and A.V. Lamsweerde (1997), "Analogical Reuse of Requirements Frameworks," In *Proceedings of the 3rd IEEE International Symposium on Requirements Engineering*, RE '97, IEEE Computer Society Press, Los Alamitos, CA, pp. 26–37.

Matsumoto, Y. (1993), "A Software Factory: An Overall Approach to Software Production," In *Proceedings of the ITT Workshop on Reusability in Programming*, IEEE Computer Society Press, Newport, RI, reprinted in: *Tutorial Software Reusability*, P. Freeman, Ed.

McCune, W. (1994), *Otter3.0 Reference Manual and Guide*, ARGONE National Laboratory: Mathematics and Computer Science Division.

Mili, H., E. AhKi, R. Godin and H. Mcheick (1997), "Another Nail to the Coffin of Faceted Controlled-Vocabulary Component Classification and Retrieval," In *Proceedings of the Symposium on Software Reusability*, SSR '97, *ACM Software Engineering Notes 22*, 3, 89–98.

Mili, H., F. Mili and A. Mili (1995), "Reusing Software: Issues and Research Directions," *IEEE Transactions on Software Engineering 21*, 6, 528–562.

Mili, R. (1996), "Assessing the Reuse Worthiness of a Component: Empirical and Analytical Approaches," Technical Report, Department of Computer Science, University of Ottawa, Ottawa, Ontario, Canada.

Mili, A., R. Mili and R. Mittermeir (1994), "Storing and Retrieving Software Component: A Refinement-Based Approach," In *Proceedings of the 16th International Conference on Software Engineering*, IEEE Computer Society Press, Los Alamitos, CA, pp. 91–100.

Mittermeir, R.T. (1998), "Hypertext: Werkzeug? – Denkzeug?" In *Electronic Networking and the Philosophy of Culture*, Ch. Nyiri and P. Fleissner, Eds., Studienverlag, Innsbruck, Austria.

Mittermeir, R.T. and W. Rossak (1987), "Software Bases and Software Archives, Alternatives to Support Software Reuse," In *Proceedings of the Fall Joint Computer Conference '87*, IEEE Computer Society Press, Los Alamitos, CA, pp. 21–28.

Mittermeir, R.T. and W. Rossak (1990), "Reusability," In *Modern Software Engineering*, P.N. Peter and R.T. Yeh, Eds., Van Nostrand Reinhold, New York, NY, pp. 205–235.

Mittermeir, R.T. and L. Wuerfl (1995), "Abstract Visualization of Software: A Basis for a Complex Hash-Key," In *Advances in Intelligent Computing*, B. Bouchon-Meunier, R.R. Yager and L.A. Zadeh, Eds., LNCS 945, Springer, Berlin, Germany, pp. 545–554.

Moineau, T. and M.C. Gaudel (1991), "Software Reusability through Formal Specifications," In *Proceedings of the First International Workshop on Software Reusability*, SWT Memo Nr. 57, Universität Dortmund, Dortmund, Germany, pp. 202–212.

Nelson, M.L. and T. Poulis (1995), "The Class Storage and Retrieval System: Enhancing Reusability in Object-Oriented Systems," *ACM OOPS Messenger 6*, 2, 28–36.

Ogden, W.F., M. Sitaraman, B.W. Weide and S.H. Zweben (1994), "The RESOLVE Framework and Discipline – A Research Synopsis," *ACM Software Engineering Notes 19*, 4, 23–28.

Ostertag, E., J. Hendler, R. Prieto-Diaz and C. Braun (1992), "Computing Similarity in a Reuse Library System: An AI-Based Approach," *ACM Transactions on Software Engineering and Methodology 1*, 3, 205–228.

Park, Y. and P. Bai (1997), "Generating Samples for Component Retrieval by Execution," Technical Report, University of Windsor, Windsor, Ontario, Canada.

Paul, S. and A. Prakash (1994), "A Framework for Source Code Search Using Program Patterns," *IEEE Transactions on Software Engineering SE-20*, 6, 462–475.

Penix, J. and P. Alexander (1995), "Design Representation for Automating Software Component Reuse," In *Proceedings of the 1st International Workshop on Knowledge Based Systems for the (Re)Use of Software Libraries*, INRIA, Sophia Antipolis, France.

Penix, J. and P. Alexander (1996), "Efficient Specification Based Component Retrieval," Technical Report, Knowledge Based Software Engineering Laboratory, Department of Electrical and Computer Engineering, University of Cincinnati, Cincinnati, OH.

Penix, J., P. Alexander and K. Havelund (1997), "Declarative Specification of Software Architectures," In *Proceedings of the 12th IEEE International Automated Software Engineering Conference*, IEEE Computer Society Press, Los Alamitos, CA, pp. 201–208.

Perry, D.E. (1989), "The Inscape Environment," In *Proceedings of the 11th International Conference on Software Engineering*, IEEE Computer Society Press, Los Alamitos, CA, pp. 2–12.

Perry, D.E. and S.S. Popovich (1993), "Inquire: Predicate-Based Use and Reuse," In *Proceedings of the 8th Knowledge Based Software Engineering Conference*, KBSE '93, IEEE Computer Society Press, Los Alamitos, CA, pp. 144–151.

Podgurski, A. and L. Pierce (1992), "Behaviour Sampling: A Technique for Automated Retrieval of Reusable Components," In *Proceedings of the 14th International Conference on Software Engineering*, ACM Press, New York, NY, pp. 300–304.

Podgurski, A. and L. Pierce (1993), "Retrieving Reusable Software by Sampling Behavior," *ACM Transactions on Software Engineering and Methodology 2*, 3, 286–303.

Potts, C. and W.C. Newstetter (1997), "Naturalistic Inquiry and Requirements Engineering: Reconciling Their Theoretical Foundations," In *Proceedings of the 3rd IEEE International Symposium on Requirements Engineering*, RE '97, IEEE Computer Society Press, Los Alamitos, CA, pp. 118–127.

Poulin, J. and K.J. Werkman (1995), "Melding Structured Abstracts and the World Wide Web for Retrieval of Reusable Components," In *Proceedings of the ACM SIGSOFT Symposium on Software Reuse*, SSR '95, ACM Press, New York, NY, pp. 160–168.

Prieto-Diaz, R. (1985), "A Software Classification Scheme," PhD Thesis, Department of Computer and Information Science, University of California at Irvine, Irvine, CA.

Prieto-Diaz, R. (1990), "Classification of Reusable Modules," In *Software Reusability, Volume I: Concepts and Models*, T.J. Biggerstaff and A.J. Perlis, Eds., Frontier Series, ACM Press/Addison-Wesley, Reading, MA.

Prieto-Diaz, R. (1991), "Implementing Faceted Classification for Software Reuse," *Communications of the ACM 34*, 5, 88–97.

Prieto-Diaz, R. and P. Freeman (1987), "Classifying Software for Reusability," *IEEE Software 4*, 1, 6–16.

Reubenstein, H.B. and R.C. Waters (1991), "The Requirements' Apprentice: Automated Assistance for Requirements Acquisition," *IEEE Transactions on Software Engineering 17*, 3, 226–240.

Rich, C. and H.E. Schrobe (1976), "An Initial Report on Lisp's Programmer's Apprentice," Technical Report MIT/AI/TR-354, Massachusetts Institute of Technology, Cambridge, MA.

Rich, C. and H.E. Schrobe (1978), "An Initial Report on Lisp's Programmer's Apprentice," *IEEE Transactions on Software Engineering 4*, 11.

Rich, C. and R.C. Waters (1988), "The Programmer's Apprentice: A Research Overview," *IEEE Computer 21*, 11, 10–25.

Rich, C. and R.C. Waters (1990), *The Programmer's Apprentice*, Frontier Series, ACM Press/Addison-Wesley, Reading, MA.

Rittri, M. (1989), "Using Types as Search Keys in Function Libraries," *The Journal of Functional Programming 1*, 1.

Rittri, M. (1992), "Retrieving Library Identifiers via Equational Matching of Types," Technical Report 65, Programming Methodology Group, Department of Computer Science, Chalmers University of Technology and University of Geteborg, Geteborg, Sweden.

Rollins, E.J. and J. Wing (1991), "Specifications as Search Keys for Software Libraries," In *Proceedings of the 8th International Conference on Logic Programming*, ICLP '91, MIT Press, Cambridge, MA, pp. 173–187.

Rossak, W. and R.T. Mittermeir (1989), "A DBMS-Based Repository for Reusable Software Components," In *Proceedings of the 2nd International Workshop Software Engineering and Its Applications*, EC2, Toulouse, France, pp. 501–518.

Rosson, M.B. and J.M. Carrol (1996), "The Reuse of Uses in Smalltalk Programming," *ACM Transactions on Computer–Human Interaction 3*, 3, 219–253.

Runciman, C. and I. Toyn (1989), "Retrieving Reusable Software Components by Polymorphic Type," In *Proceedings of the 4th International Conference on Functional Programming Languages and Computer Architectures*, ACM Press, New York, NY, pp. 166–173.

Salton, G. (1975), *Information and Library Processing*, Prentice-Hall, Englewood Cliffs, NJ.

Salton, G. and M.J. McGill (1983), *Introduction to Modern Information Retrieval*, McGraw-Hill, New York, NY.

Schumann, J. and B. Fischer (1997), "NORA/HAMMR: Making Deduction-Based Software Component Retrieval Practical," In *Proceedings of the 12th IEEE International Automated Software Engineering Conference*, IEEE Computer Society Press, Los Alamitos, CA, pp. 246–254.

Spanoudakis, G. and P. Constantopoulos (1993), "Similarity for Analogical Software Reuse: A Conceptual Modelling Approach," In *Proceedings of the CAiSE '93*, LNCS 685, Springer, Berlin, Germany.

Spanoudakis, G. and P. Constantopoulos (1994), "Measuring Similarity between Software Artifacts," In *Proceedings of the 6th International Conference on Software Engineering and Knowledge Engineering*, SEKE '94, Knowledge Systems Institute, Skokie, IL.

Steigerwald, R.A. (1992), "Reusable Component Retrieval with Formal Specifications," In *Proceedings of the 5th Annual Workshop on Software Reuse*, WISR-5, Palo Alto, CA, `ftp://gandalf.umcs.maine.edu/pub/WISR/wisr5/proceedings/`.

Sutcliffe, A.G. and N.A.M. Maiden (1994), "Domain Modeling for Reuse," In *Proceedings of the 3rd International Conference on Software Reuse*, IEEE Computer Society Press, Los Alamitos, CA, pp. 169–177.

Talens, G., D. Boulanger, I. Dedun and S. Commeau (1997), "A Proposal of Retrieval and Classification Method for a Case Library Reuse," In *Proceedings of the 9th International Conference on Software Engineering and Knowledge Engineering*, SEKE '97, Knowledge Systems Institute, Skokie, IL, pp. 502–511.

Tracz, W. (1988), "Software Reuse Myths," *ACM Software Engineering Notes 13*, 1, 38–43.

Van Rijsbergen, C.J. (1979), *Information Retrieval*, 2nd Edition, Butterworth and Co., London, UK.

Waters, R.C. (1981), "The Programmer's Apprentice: A Session with KBEmacs," *IEEE Transactions on Software Engineering 7*, 11, 1296–1320.

Waters, R.C. (1982), "The Programmer's Apprentice: Knowledge-Based Program Editing," *IEEE Transactions on Software Engineering 8*, 1, 1–12.

Wood, M. and I. Sommerville (1988), "An Information System for Software Components," *ACM SIGIR Forum 22*, 3, 11–25.

Zaremski, A.M. and J.M. Wing (1993), "Signature Matching: A Key to Reuse," In *Proceedings of the*

SIGSOFT '93: ACM SIGSOFT Symposium on the Foundations of Software Engineering, Software Engineering Notes 18, 5, 182–190.

Zaremski, A.M. and J.M. Wing (1995a), "Signature Matching: A Tool for Using Software Libraries," *ACM Transactions on Software Engineering and Methodology 4*, 2, 146–170.

Zaremski, A.M. and J.M. Wing (1995b), "Specification Matching of Software Components," In *Proceedings of the SIGSOFT '95: 3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering, Software Engineering Notes 20*, 4, 6–17.