Uncertainty Aspects in Component Retrieval

R. T. Mittermeir,
H. Pozewaunig
Institut für Informatik-Systeme,
Universität Klagenfurt,
Austria
{roland, hepo}@ifi.uni-klu.ac.at

A. Mili
Dept. of Computer Science
and Electrical Engineering
West Virginia University,
Morgantown, WV 26506
amili@cs,wvu.edu

R. Mili
School of Engineering
and Computer Science
University of Texas at Dallas
Richardson, TX 75083 USA
rmili@utdallas.edu

Abstract

Successful software reuse depends on many factors, adequate description of reusable software is one of them.

This paper focuses on some of the inherent problems in adequately describing software for later focussed retrieval. Out of these considerations, a hybrid approach, combining well known techniques from library science with techniques based on the inherent property of software, its executability, is presented.

1 Introduction - Motivation

Software reuse has many facets and several motivators and inhibitors [22, 23] have been discussed in the literature. Some leading figures derive from these lists of inhibitors, even that software reuse is just a social and organizational problem and technical solutions should be held back before those issues are solved.

We do not buy into such advice, as technology is intermittingly coupled with the social field in which it is applied. Hence, what might be considered inappropriate, given some technology might well be acceptable on the basis of a different (not necessarily better) technology. Further, we point out that the field has changed within the last decade. On one hand, the discussion of (object-oriented) patterns and frameworks has brought new substance to the discussion of software reuse [10]. On the other hand, the quick adaption of the internet as provider of various information, amongst them as provider of reusable software [1, 4] brought new motivation for re-considering schemes to support component based reuse. Overviews of various attempts for reusing software can be found in [7, 11, 13].

Here, we focus on the basics of the software-retrieval (and by the reverse side of the same token: the software-description) issue. In section 2, we will address the problems involved with description of information on a more compact, and hence more abstract, general level. We will see that this leads to uncertainty problems, where different persons at different times are required to make identical abstractions to lead to an adequate match. In section 4 we will consider a distinguishing factor between software and other forms of information, the fact that software is executable. This will open up new avenues for software description, avenues that are almost orthogonal to conventional attempts to describe the semantics of software assets. Based on these considerations, we discuss various approaches that benefit from the executability of software, the context in which these apply; finally, we sketch several approaches to render these multidimensionality operationally.

To arrive at this point, we first discuss the semantic problems resulting from matching descriptors while one actually is looking for matching concepts. Based on these reflections, various approaches to describe software are considered and uncertainty problems related to descriptive, operational, and structural approaches are discussed. As consequence, we propose a hybrid approach, combining the concepts discussed so far with aspects developed in the field of software testing. The paper concludes by discussing some open problems and further extensions of this approach.

2 Issues of Uncertainty

We refer to the well known correspondence between the information retrieval problem in general and software retrieval in particular [5, 6, 12]. Thus, we acknowledge the fact that a query into a repository of software components will in general yield a set of components that more or less qualify with respect to the intent of the requester, while another set of component that might also (more or less) qualify are not retrieved.

This "more or less qualified" refers to the fact that the set of components qualifying with respect to the developers needs is actually a fuzzy set. The fuzziness of this set is hidden by the specific interpretation of the query. Whether this query is rather restrictive or rather open specifies in fact the level of the α-cut.

Information retrieval literature uses different terminology. The membership function of a component is not quantitatively formalized for each element of the repository. Instead, the query is qualified with respect to

- *recall*, the ratio of relevant retrieved components relative to the number of relevant components held in the repository, and
- *precision*, the ratio of retrieved relevant components relative to the total number of retrieved components.

Before further elaborating on this relationship between fuzzy membership in a set of qualifying components and the standard quality indicators for information retrieval, we list some premises of our work:

- *reuse in the large:* We address the issue of software reuse across project-team and across organizational boundaries. This implies that the individual reuser has little or no tacit knowledge about the context and purpose in and for which a component has been developed.
- *abundance of components:* As we consider reuse across organizational boundaries, it seems fair to assume that there is an abundance of (even relatively specialized) components available. Whether these will stem from some component vendor or from some (managed) repositories distributed over the internet will not be discussed further in this paper.
- *dominance of precision:* Literature on software retrieval is to a large extent biased towards high recall. With small repositories, this seems justified. The reuser has first to find one or two components potentially qualifying; to check, whether they really qualify can be done later off line. With the assumption of an abundance of components one can almost be sure that several "more or less" qualifying components are available and that one will get several of them. However, one does not want to invest too much time in off line verification (testing!?) whether the components yielded will really satisfy the intended purpose. Hence, precision becomes the dominant criterion.
- *library analogy:* The above premises match well to the premises a manager of a large library (of books) has when searching for some information. One is almost sure, the library contains books where this information can be found and several books might provide the right answer, albeit at different degrees of intellectual accessibility. The key question is how to find the book where the sought information is contained in a proper form.

2.1 Information Retrieval in a Library

It seems that the discussion of software reuse is laden with so many implicit arguments and hidden hopes that we first present our arguments in the context of using a classical library. The analogy to retrieving reusable components from a software repository will immediately become obvious.

If we go to a library, we are in general not interested in a particular book. We are interested in the specific information contained in this book. Likewise with software, we are generally not interested in a particular piece of software but in the functionality (in the broadest sense of this word, encompassing also various performance, security, ... considerations). With the book (assuming you are looking for a scientific book) the presentation (order, font, even arrangement of the line of arguments) might be immaterial. Likewise, the particular algorithms used in the software might be immaterial. Both arguments hold, as long as the desired degree of precision (of the information described resp. the input-output mappings performed) is attained.

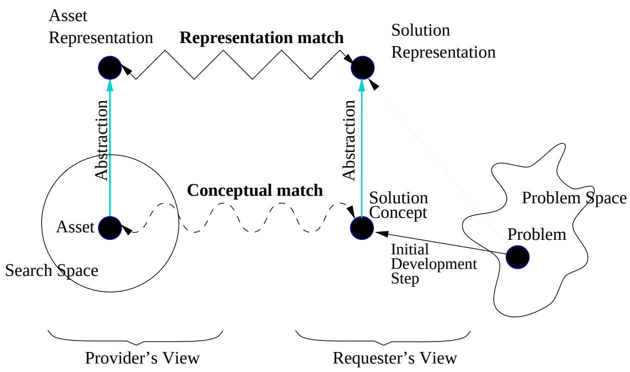

Figure 1: Mapping Problem-Needs to available Assets

Thus, we see that there is essentially a twofold abstraction process. One from the desired service to the artifact (asset) that promises to provide this service; the other from the asset providing some service to the surrogate representing or describing this asset.

2.2 Software Retrieval

Similar arguments hold with software retrieval. The reuser – in general – is not interested in a specific representational form of the component. He/She is interested in the specific functionality a component provides. Software has many commonalities with written literature, e.g. the fact that, represented as source programs, it might be considered as a text or character string (as source programs, as a specially encoded bit string). Another commonality is, that the patron of the library/repository is not interested in this character string per se. With the library, he is interested in the information conveyed by this string; with the repository, the reuser is interested in its functionality. Thus, the main difference between books and software is that software is executable.

In devising a scheme for organizing a software repository, one can benefit from this difference. While the information conveyed by a text is inaccessible to intersubjective formal reasoning or description, the functionality a software component provides can be intersubjectively described in form of the input-output mappings yielded by the component, resp. by a sample thereof. Thus, while both, literature and software, require some meta structure to facilitate access to the items contained in the library/repository, the library's meta structure has to rely on agreed upon descriptions of the works it refers to (such as author name, series, publisher, etc.) and presupposed very high level abstractions of the (information) contents of the works (keywords). The software repository, while directly benefiting from these analogies, can also provide direct representations of functionality, albeit only on the level of examples. In the next section we will discuss this topic in detail.

3 Abstractions and Uncertainty

Resolving the uncertainty issue as to whether a component retrieved is a sufficiently close proxy to the component sought based on the problem description at hand thus involves (various) abstraction steps. Certainly, these are at least the abstraction used by the (software-)librarian when integrating an asset into the repository and the abstraction used by the requester when searching or checking out a component. As long as these two abstractions are made according to identical rules of abstraction, mappings will be perfect, except that the information-loss due to the abstraction might lead from a potential $1:1$-correspondence between problem-need and available asset to a $1:n$-correspondence. Many cases will not be so constrained that a direct relation between problem space and search space will lead to a $1:1$-correspondence. Several, say m, components might be adequate choices. Hence, assuming an ideal abstraction process on both sides and several fully qualifying components on the repository side, we will arrive in the worst case at a $1:m*n$ correspondence. In other words, out of $m*n$ components retrieved, m components will qualify, the remaining $m*(n-1)$ are in the solution of the query only due to the querie's inherent ambiguity.

However, the assumption of fully corresponding abstractions is unrealistic for two reasons: First, the librarian's and the requester's abstraction are made at different times and in a different contexts. Therefore, they will be subject to different biases. Second, one can formulate hard and fast rules only for some abstraction-dimensions. For others – unfortunately, for the more interesting ones – only relatively vague rules can be formulated. To explain this in detail, we consider again the case of a classical library.

3.1 Abstractions used for classifying books

The success of information retrieval in general depends largely on the adequacy of the abstraction process used when describing the respective artifact once it has been incorporated into the repository. Librarians undergo a formal training to ensure consistent classification of the assets they are guarding. As far as this training is concerned, the adequate and intersubjectively consistent description of properties of the physical artifact containing the respective information, is quite advisable (page numbers, name(s) of author(s), standardized capture of the book-title, ISBN or LoC number, etc). However, in recognition of the fact that readers are not looking for books but for the information contained in those books, their attempts to bridge this gap and describe also the contents by standardized means are less successful.

3.2 Abstractions used for classifying software

In light of the problems just described in classical library science, it seems fair to acknowledge the same limits for software description, a field that has not been that deeply explored and where descriptions are just made by classification amateurs, i. e. in most cases by software developers.

The problem with content based classification is that the classification context is different for each requester and even more so for the librarian (resp. the developer of the software). The reader, or software developer seeking reuse, has a question stemming from some particular application. The librarian does not have this context. He/she is neutral with respect of the artifact. Her training will allow her to extract some keywords from the title of the book (or in more elaborate cases from parts of the contents, but we may not assume that she has read the full book before classifying it). This passes the burden of classification or of giving proper hints to the author and there the library example and the software repository converge again. The author of the book might have had a set of complex criteria for choosing a title, marketing criteria not being the least important one. The author of a piece of software will be biased by the application this software has been written for.

Thus, we notice that descriptive (keyword based or faceted) schemes are dependent on at least shared criteria for describing artifacts. These criteria can be assumed within large projects (shared application terminology) or in carefully restricted domains (service functions). These are also the domains, where software reuse as been successfully practiced in the past (e.g. [20]). When both, the criteria of shared abstractions due to shared work (project) or shared abstractions due to shared concepts (service functions) fail to hold, keyword based abstractions become of limited use. The recommendation, not to seek software reuse across domains might be a consequence. Considering anonymous reuse across the internet will have little chance if keyword based descriptions are the only mechanisms to capture the semantics of available components. Potential reusers will be drawn away from the search, since the ratio between recall and precision will be so poor that the effort to verify whether the components retrieved (or which of the components retrieved) satisfies the needs will not warrant the search effort (for a more detailed discussion, see [14]).

3.3 Strata of Abstraction for Software Description

Referring back to the observation that the content of software can be described by its input-output behavior, and recognizing, that this input-output behavior can be described in terms of a formal specification or in terms of (exemplary) tuples of input-output values, we come to the following layers of abstraction for software description:

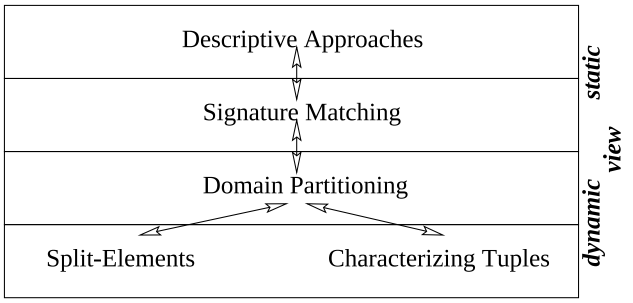

Figure 2: Different Strata of Abstraction

1. *Textual Descriptions (Keywords* and *Facets)* provide a high level description of the semantics of a component. With facets, this description is structured according to preconceived categories. Thus, chances to arrive at matching abstractions for matching concepts will be improved. Nevertheless, these abstractions rely basically on human intuition. To perform them automatically has been attempted [8]. But these attempts soon reach their limits [12].

2. *Signature Matching* [15, 24, 25], on the contrary, is rather a formal approach, benefiting from the fact that software, even if considered as text, is a highly structured text. Basically, it provides a description on the interface level. If performed naively, one would arrive at a match according to the syntax of the interface a component provides. This, however, will be too constrained. Hence, various schemes have been proposed to get rid of irrelevant syntactical constraints, such as order of parameters, composition of aggregate types, subtyping etc. With these relaxations, however, syntactical information is lost. What remains is just an attempt to use types as indicators for the semantics of the components involved.

3. *Domain Partitioning* has, to our knowledge, not been used so far for software retrieval purposes. It might be considered a natural extension though of signature matching, as it enriches the description of the semantic content of a component only vaguely pointed to by the signature, but still spares much of the effort involved in fully describing the semantics of a component by some formal specification. Thus, it can be considered as simplified version of the layered specification matching proposed in [15].

4. *Characteristic Tuples,* finally will avoid the abstraction involved in domain partitioning. By providing concrete examples of input-output values processed by the component at hand (needed by the problem in question), requesters can directly see, to which extent some component will satisfy their needs.

5. *Split Elements* are another way to describe components by means of example values. The approach is more involved than the one asking just for characteristic elements, since here, tuples indicating extreme conditions are required. The motivation to demand them, stem from software quality assurance, specifically from the testing strategy of boundary value analysis.

The first two strata do not consider the special nature of software as executable text. The third strata draws indirectly on the executability by providing an incomplete specification including also some elements of pragmatism. The fourth strata, represented above by the fourth and fifth item, indicate partial semantics and bank fully on the operational characteristics of software. Operational approaches to software retrieval have been proposed already by several authors [18, 19, 9]. Their arguments are slightly different though from the ones we present in the next section.

3.4 Uncertainty Management in the individual Abstraction Strata

3.4.1 Keyword-based Abstractions

With keyword based abstractions as representatives of "descriptive methods", one fully relies on human individuals using the same frame of reference to coin abstractions. Without this assumption, keyword based approaches would be highly ridiculous, since in principle, both, the solution sought as well as the available assets have to be abstracted into (sets of) string(s) of the length k. Assuming an alphabet of magnitude a, a^k such strings can be formed. Thus, without such a common frame of reference, the chance of a successful match would be as small as $\frac{1}{a^k}$.

Apparently, in practical cases, one arrives at much higher match probability. The improvement is due to the alignment of a frame of reference by common terminology within a project, a company, or even a particular domain. This assumption is usually made implicitly. Justification cannot be assessed without knowing the specific context and without empirical analysis. Hence, we cannot give any quantitative arguments here. We can say though, that the deficiency to answer quantitatively to this question can be cited as a reason, why one cannot directly apply fuzzy reasoning to the software/information retrieval problem by adjusting precision simply by raising the level for the α-cut referred to in the introduction of this paper.

3.4.2 Signature Abstractions

We consider an ordered tuple of *identifier* $\times$ *type* pairs as signature of the component. In this tuple, however, the actual identifier used to denote a particular parameter is as irrelevant as the particular order in which the parameters are listed in the signature. What matters, though, is whether a parameter serves as input or output argument of the procedure. Hence, the available information is in essence two bags of types, a bag of input types and a bag of output types (assuming that in-out parameters figure in both bags).

Taking the illustrative example of a function f, mapping from a pair of integers to an integer, one sees how little information is provided. What we have for sure is the information that this function maps from $|integer|^2$ to $|integer|$. Thus, again, the probability that some $integer \times integer \Rightarrow integer$ function found will be the function sought, can be computed.

Again, this seems to be very little information. However, it is free from the contextual information needed with descriptive approaches. There is, however, a design bias, that is difficult to deal with in the context of signature matching. As long as it involves complex user defined type aggregations (record structures), dissolving these structures and looking for equivalence at the leaf-level will solve the problems. With object-oriented subtyping, optional parameters, and generic parameters, arriving at correct matches will be much more challenging [24, 25, 16].

As a practical observation one might mention though, that the uncertainties involved with descriptive approaches and the uncertainties involved with signature matching seem to be almost orthogonal. Hence, their combination seems advisable.

3.4.3 Domain Partitioning

The arguments to be raised here follow directly from the arguments raised with signature matching. As domains are partitioned, we no longer have the product of the cardinality of the types involved as basic uncertainty to cope with, but only the sum of the product of those subsets of the types involved, falling within a particular domain partition. Depending on the specifics of the software and the complexity of the individual partitions, this can become relatively fine grained. Hence, we refrain from indicating a particular formula for this quantum. The general line of reasoning and the hint to the combinatorial nature of the problem should suffice to demonstrate that relative to plain signature matching, domain partitioning affords us a dramatic reduction of uncertainty.

To obtain full benefit of domain partitioning, one does not necessarily arrive at extremely fine grained partitions. As key step, it might suffice to differ between the domain of a function and its "from-set" [3]. This simply partitions values in the Cartesian product of types of input parameters into the set, where a defined result will be obtained and the one, where no result is obtained. The first set can be partitioned further into values where meaningful results will be obtained, and in those, where only error messages will result.

3.4.4 Example Values

At first sight, providing example values seems weak. This may be inferred from the limitations that programming by example [21] ran into. In comparison to and in extension of the (established) approaches, it becomes interesting though. This holds especially, if retrieval by example is seen not as exclusive approach but as complementary to the ones discussed so far.

It is further in line with the arguments that partial specifications are sufficient for software retrieval [15]. Example values provide just such a partial specification. However, their partiality is of different nature from the partiality a specification has, in which certain attributes rely on context dependent interpretation.

To assess the partiality of a specification based on example values, we have to conceive of the function or procedure described in its extensional form as a huge (possibly infinite) table of input-output mappings. The example values indicated are a sample drawn from this table. Their quality is dependent on the sampling strategy used.

Podgurski and Pierce [18] recommend the operational profile as adequate sampling strategy. We strive rather at a different strategy stemming from domain partitioning. There are several arguments for this different proposal. One among them is, that the operational profile is context dependent (and may shift over time), while domain partitions considered by a component are inherent in both, the problem and its solution and therefore stable. Hence, values defining the boundaries of the subdomains involved, seem to be stronger discriminators. They have also benefits concerning matching abstractions, because librarian and requester will follow implicitly similar rules to arrive at the same (or equivalent) boundary values.

The rationale for the latter claim is given below. If, for whatever reason, one does not buy into this rationale, characteristic values from a domain might be used. Since they will be tool-generated, consistency can be achieved relatively easily on the librarians side. To achieve it also at the requesters side might be more difficult. Here however, we refer to the fact, that such values can be used for two different modes of operation during software retrieval. One is the classical way, that the repository waits for the requesters query and reports the assets with descriptions satisfying this query (*searching*). The other is, that, after the set of candidate assets has been already narrowed down by some other form of queries, the repository provides descriptions for those candidates still qualifying for the requesters inspection (*browsing*). Here, no prefacto synchronization between the librarian's and the request-or's abstractions is needed. Counter examples will play a specific discriminatory role in the mode.

3.5 QA in the Reuse Process

In order to assess both feasibility and desirability of the approach outlined above, we briefly look into the respective processes of the asset provider and the asset requester. In both cases, we assume that the respective individuals or teams follow a high quality process.

3.5.1 Development For Reuse

As we are not interested in the particular syntactic shape of a reusable component, we are also not very interested in the details of its development. However, we have to assume that the final executable form of the component is faithful with respect to its description. To assure such faithfulness, the component must have been adequately tested. Because the "description" against which the component is tested has nothing to do with its internal structure, such tests must have been black box tests. In order to do them, the developer needs both:

- A specification indicating at least the most critical values it has to be tested against. These values will result from domain partitioning [17, 2]
- The test suite used during testing / for arriving at this conclusion.

To include both items with the code of the component into the repository seems not to require any additional effort on the developer's side.

3.5.2 Development With Reuse

For the requester, we assume that he developed some solution concept during the initial development step. We will not make any assumptions as to how this initial development step is performed, nor in what form the solution concept (see Figure 1) is expressed. However, we might assume that the developer with reuse ascertains that the retrieved component really satisfies the needs expressed by the problem and captured in the solution concept. – This, again will be done by testing.

Whether this testing is done off line or whether it is done (partly) already during the retrieval process will be immaterial from the requester's perspective. The only key problem might be, how the requester arrives at the suite of test data to be run against the component. Domain partitioning might be one choice, getting stimulated by the suite of test data provided by the builder of the component might be another one that reduces quite a bit of effort.

4 A Hybrid Approach, resting on Operational Properties

Based on the considerations raised up to now, we propose a layered approach to software description:

- Textual description, in spite of its theoretical weakness, serves to initially narrow down the search space in the repository. Thus, due to its practical relevance, we use it for coarse grain discrimination among components.

- Signature matching is used on the next level. The arguments for signature matching are (1), that it is relatively inexpensive, and (2) that we need the techniques from signature matching in order to perform the next step:

- Matching of input-output to make the final choice.

Textual descriptions (keywords, facets) and signature matching need no further explanation, since they are sufficiently discussed in the literature. Hence we focus in the rest of the paper on the match between input-output tuples, knowing that the order indicated above is to be seen as default strategy that can be broken, whenever intermediate results suggest to do so. We do so, by responding to a set of critical questions:

Do tuples adequately describe the semantics of components? We consider this to be the wrong question. Obviously, one can construct arbitrary counter examples. However, they are unbiased descriptions and they are descriptions that can be easily refined just by adding more tuples. Which tuples are worth to be included follows from the theory of black box testing. *How should the requester "guess" the correct tuples?* As explained above, domain partitioning will serve as a strong clue that at least some tuples in the librarians description and some tuples in the requesters one are identical (modulo order of parameters). The problem of the order of parameters is dealt with by techniques related to signature matching. *Does the requester need to "guess" the correct tuples?* Not really! We consider several modes of operation of this approach:

- Static matching of tuples: Here, the requesters tuples are matched against the librarians tuples. In this case, "correct guesses" are important.

- Dynamic matching: Here, the requester provides input and the retrieval system executes components preselected by previous steps with these inputs. The requester inspects the resulting output. Hence, no guessing of identical tuples is needed.

- Browsing: Here, the retrieval system prompts the requester with tuples stored with (or computed by) the preselected components. We want to point particularly to the fact that the professionalism of the developer of the reusable components is directly paid back in the system developed with reuse (Note the power of good "counter-examples").

What about complex types? This is a classical problem of signature matching. Since the operational approach has to use concrete values anyway, different abstractions that might be used on the type levels no longer matter. *What about optional parameters? What about generic parameters?* This, again, is a problem dealt with in signature matching approaches. It can be seen in a new light here since it matters particularly with highly generic components made explicitly for reuse: Consider e.g. the ANSI C-function `qsort`. Simple minded signature matches would have problems in this case. Textual descriptions would retrieve it though. Since the hybrid approach is in principle stratified, one can always go back to a previously selected richer pool of candidates and choose among them. In the case of highly default-parameterized functions or functions with strong generic parameters (or curried functions, to mention also the other extreme) matches over projections of the tuple-space will point into the right direction. Such projections are much stronger as in plain signature matches, since one has not just a few types as match arguments, but a set of tuples, that may be characteristic due to the composition of tuple-projections remaining. *Feasibility of Implementation?* A full discussion of the implementation is beyond the scope of this paper. We can indicate here only, that for side-effect free procedures using only static types hashing techniques and internal canonical orders can buffer most of the divergences one has to deal with. *Streams and Interaction* are not dealt with in our initial version. Addressing them will be subject of future research.

5 Conclusion

The general semantic problem of retrieving artifacts for their content and not for their external appearance has been discussed and related to software reuse and software retrieval. As consequence of these considerations, a hybrid scheme was proposed, building on the combination of the strength of descriptive, structural, and operational techniques. The basic message of the paper, though, is not the particular combination of these approaches but the specifics of the operational approach proposed here, combining ideas from software retrieval with ideas from software testing.

Here, the approach has been discussed only on the basis of transient functions. This restriction stems from the specifics of the input-output examples given. An extension to state-based software as well as to interactive software is subject to further work.

References

[1] G. Arango. Software Reusability and the Inter-

net. In M. Samadzadeh and M. Zand, editors, *Proc. SSR '95, ACM SIGSOFT Symposium on Software Reusability*, New York, N.Y., April 1995. ACM Press.

[2] B. Beizer. *Black-Box Testing.* John Wiley & Sons, Inc., New York, N.Y., 1995.

[3] A. Diller. *Z - An Introduction to Formal Methods.* John Wiley & Sons, Chichester, England, 2nd edition, 1994.

[4] S. Fickas. Workshop Software on Demand: Issues for Requirements Engineering. In J. Mylopoulos, editor, *Proc. 3rd IEEE International Symposium on Requirements Engineering*, RE '97, pages 222 – 223, Los Alamitos, CA, January 1997.

[5] W.B. Frakes and R. Baeza-Yates, editors. *Information Retrieval: Data Structures and Algorithms.* Prentice Hall, Upper Saddle River, N.J., 1992.

[6] W.B. Frakes and P.B. Gandel. Representing Reusable Software. *Information and Software Technology*, 32(10):653 – 664, December 1990.

[7] W.B. Frakes and C. Terry. Software Reuse: Metrics and Models. *ACM Computing Surveys*, 28(2):415 – 435, June 1996.

[8] R. Girardi and B. Ibrahim. Automatic Indexing of Software Artifacts. In W.B. Frakes, editor, *Proc. 3rd IEEE International Conference on Software Reuse*, pages 24–32, Los Alamitos, CA, November 1994.

[9] R.J. Hall. Generalized behaviour-based retrieval. In *Proc. 15th International Conference on Software Engineering*, ICSE 93, pages 371 – 380, Baltimore, MD, May 1993. IEEE Computer Society.

[10] R. Johnson. Components, Frameworks, Patterns. In M. Harandi, editor, *Proc. ACM SIGSOFT Symposium on Software Reusability, SSR '97*, volume ACM SEN 22(3), pages 10 – 17, New York, N.Y, May 1997. ACM Press.

[11] Ch.W. Krueger. Software Reuse. *ACM Computing Surveys*, 24(2):131–183, June 1992.

[12] A. Mili, R. Mili, and R.T. Mittermeir. A Survey of Software Reuse Libraries. *Annals of Software Engineering*, to appear, 1998.

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

[14] R. Mili and R.T. Mittermeir. Ex-ante Assessing of Reusability. In Mayr Györkös, Krisper, editor, *Proc. 4th Conference on Re-Technologies for Information Systems, ReTIS '95*, number 80 in OCG Schriftenreihe, pages 41 – 45, 1995.

[15] R.T. Mittermeir and E. Kofler. Layered specifications to support reusability and integrability. *Journal of Systems Integration*, 3(3):273–302, September 1993.

[16] T. Miura and I. Shioya. On Complex Type Hierarchy. In *IEEE Knowledge & Data Engineering Exchange Workshop*, pages 156 – 164, Newport Beach, California, November 1997.

[17] G.J. Myers. *The Art of Software Testing.* John Wiley & Sons, New York, N.Y., 1979.

[18] A. Podgurski and L. Pierce. Behaviour Sampling: A Technique for Automated Retrieval of Reusable Components. In *Proc. 14th International Conference on Software Engineering*, 14, pages 300 – 304, Melbourne, Australia, May 1992. IEEE CSP.

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

[20] R. Prieto-Diàz. Implementing Faceted Classification for Software Reuse (Experience Report). In *Proc. 12th International Conference on Software Engineering*, ICSE, pages 300 – 304, Nice, France, March 1990. IEEE CSP.

[21] P. D. Summers. A Methodology for LISP Program Construction from Examples. *Journal of the ACM*, 24(1):161 – 175, January 1997.

[22] W. Tracz. Software Reuse: Motivators and Inhibitors. In *Proc. COMPCON S'87*, pages 358–363, 1987.

[23] W. Tracz. *Collected Confessions of a Used Program Salesman: Institutionalizing Software Reuse.* Addison-Wesley, Reading, MA, 1995.

[24] A.M. Zaremski and J.M. Wing. Signature matching: a tool for using software libraries. *ACM Transactions on Software Engineering and Methodology*, 4(2):146 – 170, April 1995.

[25] A.M. Zaremski and J.M. Wing. Specification matching of software components. *ACM Transactions on Software Engineering and Methodology*, 6(4):333 – 369, October 1997.