A Practical Approach to Web Service Discovery and Retrieval

Conference Paper · August 2007

DOI: 10.1109/ICWS.2007.12 · Source: IEEE Xplore

Colin Atkinson
Universität Mannheim
212 PUBLICATIONS **5,724** CITATIONS
SEE PROFILE

Oliver Hummel
Karlsruhe Institute of Technology
56 PUBLICATIONS **704** CITATIONS
SEE PROFILE

A Practical Approach to Web Service Discovery and Retrieval

Colin Atkinson, Philipp Bostan, Oliver Hummel and Dietmar Stoll
Institute of Computer Science, University of Mannheim, 68131 Mannheim, Germany,
{atkinson|bostan|hummel|stoll}@informatik.uni-mannheim.de

Abstract

One of the fundamental pillars of the web service vision is a brokerage system that enables services to be published to a searchable repository and later retrieved by potential users. This is the basic motivation for the UDDI standard, one of the three standards underpinning current web service technology. However, this aspect of the technology has been the least successful, and the few web sites that today attempt to provide a web service brokerage facility do so using a simple cataloguing approach rather than UDDI. In this paper we analyze why the brokerage aspect of the web service vision has proven so difficult to realize in practice and outline the technical difficulties involved in setting up and maintaining useful repositories of web services. We then describe a pragmatic approach to web service brokerage based on automated indexing and discuss the required technological foundations. We also suggest some ideas for improving the existing standards to better support this approach and web service searching in general.

1. Introduction

Although web services have received a great deal of attention over the last few years, and many companies have experimented with their use, the expected use of web services as a medium for B2C and B2B interaction has failed to take off to the extent expected. Web services were also touted as a way of boosting software reuse by encouraging developers to assemble new applications from reusable parts rather than by writing everything from scratch. However, examples of serious enterprise applications that use third party web services to realize their functionality are few and far between. The vast majority of web service applications today are within, rather than between, enterprise boundaries and most web services are custom built for the purpose in hand. In effect, therefore, web services are primarily used today as a convenient middleware and wrapping technology rather than as the basis for component-based development and software reuse.

The basic problem is the failure of current technologies to successfully support the "publish and find" element of the core web service vision. As illustrated in Figure 1, which is a standard picture in most web service literature, the idea of bringing together web service providers and users via some form of brokerage service has been a core part of the web service vision right from the start. After describing the interface to their web service using WSDL, the idea is that service providers publish their services in a UDDI [11] repository by providing appropriate "meta data" such as provider identity (white pages), a categorization of the provider's industry (yellow pages) and technical information necessary to invoke the service (green pages). Developers interested in using web services are then meant to be able to find components suitable for their needs by browsing the registry or using the keyword-based UDDI search facilities.

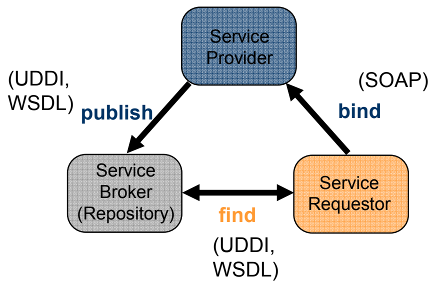

Figure 1. Standard Web Service Brokerage Model

Attempts to provide public repositories based on this vision have not been very successful, however. The most well known attempt was the so called UDDI Business Registry (UBR) supported by IBM, Microsoft and SAP, which after several years of service was quietly closed down early in 2006 because it contained only a few hundred reachable web services, and the ratio of actual web services to "junk" was very low [5]. Moreover, the few web sites that specialized in providing a repository for web services, such as xmethods.net or bindingpoint.com do not use UDDI anymore or have also recently been shut down. Instead, those remaining typically organize links to web services in a hierarchically-organized taxonomy designed for manual browsing. However, we believe the reasons for the failure of UDDI-based public repositories do not lie in the nature of the standard per se, but in the philosophy that lies behind it – namely the brokerage philosophy that relies on human maintenance of the repository and the search approach based on browsing the repository for appropriate services.

Our argument in this paper is that concepts from component-based development and component markets have been too naively transferred to web services, with the result that many of the related problems have been inherited as well. We investigate how recently emerging "code search" technologies can be adapted to provide a foundation for solving the problematic "brokerage" aspect of the web service vision. In section 2, we briefly review the history of component repository research and explain what the new generation of code search technologies add to the picture. In section 3 we then discuss the issues involved in populating a web service repository with working artifacts, and how this can be kept up to date to keep the ratio of working services to retired services reasonably high. Section 4 describes an important problem related to the different perspectives that service providers and service consumers often have of a given web service, and presents some ideas for relating these in a systematic way. Finally, in section 5 we present some suggestions for enhancing web service standards and usage conventions in order to better support effective web service brokerage mechanisms. Section 6 then concludes with some final remarks.

2. Background

The idea of software reuse based on component markets has been around for almost forty years [8], and all the evidence to date suggests that repositories whose contents are managed by humans are doomed to failure [3], [4], [10]. Typically, one of two things happens. Either,

- the quality of their contents quickly degrades and becomes unusable, or
- the overhead associated with managing and maintaining the quality of the repository becomes so large that it far outweighs the benefits.

However, as Poulin predicted in 1995 [12], the ultimate result is always the same. Once the size of a repository managed by humans goes above a certain threshold (estimated to be about 200 entries) it quickly becomes unusable in practice. Indeed, the few successful reuse approaches described in the literature (such as [6]) were based on component libraries with roughly this number of assets.

Since web services are just another form of component from a repository point of view, the overall vision of web services as a vehicle for software reuse and pan-enterprise application integration has no chance of being realized without some kind of mechanism for connecting web service developers to web service consumers. Until an effective brokerage approach is developed, therefore, web services will remain little more than a convenient middleware and wrapping technology. Fortunately, there is hope for an alternative way of brokering contacts between service providers and users. Recent advances in search engine technology coupled with the vast growth of open source software on the Internet have triggered a lot of interest in so called "code search engines" that allow users to search for freely downloadable source code. The most well known is Google Code Search, made public in July 2006, but at the time of writing there are well over a dozen code search engines available on the web.

These "engines" essentially side-step the classic "component repository problem" [16] because they do not rely on the human maintenance of content in the way described above. Instead their ability to deliver useful search results relies on the sheer volume of code available on the Internet and the ability of clever algorithms to filter out "good components" that match a user's needs. Although this is a very young technology, and the jury is still out on which of the currently available search engines provide a genuinely useful service, it seems likely that this technology will play an important part in software engineering in the future. And it also offers a potential way of side stepping the repository problems that stand in the way of effective web service brokerage. However, because web services do not have searchable code in the traditional sense, using this technology to provide a search engine for web services presents some new challenges not faced before.

2.1. The Internet as a Component Repository

The idea of using open source software from the Internet as a reuse repository is not new. Booch and Brown [2] already proposed this idea back in 2002, for example. The idea is superficially very appealing because the Internet can be viewed as a self regulating repository that requires no explicit maintenance effort. However, turning the amorphous mass of information on the Internet into a practical vehicle for software reuse depends on three fundamental building blocks –

1. the number of downloadable software assets being above a "critical mass" needed to offer a reasonable chance of finding the required functionality,
2. indexing algorithms and tools that enable a map of all the available software assets to be efficiently generated and stored,
3. search algorithms that can effectively filter out unsuitable components and identify assets that match a user's need.

It has only been within the last two years or so that the right building blocks have fallen into place and the development of useful code search engines has become possible. With the rise of Linux, Mozilla and other popular open source software millions of source code files have been made available over the Internet, and popular open source hosters such as SourceForge store well over 100,000 projects on their servers. However, none of these provides sophisticated search functionality to make this vast amount of code reusable. It is possible to search for code using clever queries to mainstream search engines such as Google and Yahoo [5], but since none of the major engines officially supports the required filetype filters and only make their programming interface available in a very limited way, this is not a viable way of supporting a serious code search facility.

The emergence of the highly efficient, open source indexing engine, Lucene [7] together with its accompanying suite of applications such as the Nutch web crawler, provided a solution to the second requirement identified above by making available the tools capable of indexing vast quantities of components. Many of the recent generation of code search engines use Lucene as their underlying indexing mechanisms. The largest four in the order of their appearance on the market are Koders.com, Krugle.com, merobase.com[1] and Google.com/codesearch (GCS). These engines demonstrate the feasibility of setting-up and maintaining indices of millions of software components as shown in the following table and offer a solution for the first two of our requirements specified above.

Table 1. Overview of component search engines

	Koders	Krugle	merobase	GCS
number of comp.	~2.5 M	~6 M	~10 M	~6 M
number of languages	37	32	48	46

The size estimates shown in the table were obtained by sampling the code engines with special queries that essentially ask for "all components in a specific language".

2.2. Searching Components and Services

Achieving the third of the above three requirements (i.e. the query algorithms) is less straightforward however, and this is one of the main areas of competition between the search engines. The simplest and most direct way of searching for particular software components in Lucene indices (or some other similar technology) is to look for a particular string in the source code as Mili and Mili [9] pointed out in their survey almost ten years ago, and all of the engines identified above support this form of search.

However, the results obtained by such a naive matching approach are often not very satisfactory because they fail to take the "meaning" or "role" of different source elements into account. Thus, a simple string-based search on the string "stack" will fail to distinguish between source code modules that are supposed to "be" a stack and those that simply "use" a stack. Any module that contains the string stack in its source code will be returned in the result set regardless of its role.

UDDI repositories [11] also share this philosophy of keyword-based searching supported by a hierarchical categorization. However, there are several fundamental weaknesses with this approach:

- as the number of indexed services grows, the categories become more generic and less useful in pin-pointing specific services,
- different users (i.e. publishers and/or consumers) often adopt different naming conventions and interpretations of concepts,
- for web service consumers, browsing through lots of categories and analyzing the capabilities (e.g. interfaces) of a service manually is a very time consuming activity,
- for web service publishers, allocating web services to appropriate categories and advertising services in an effective way poses a difficult challenge.

These problems have to a certain extent been alleviated by the introduction of new concepts in Web 2.0 like tagging which are supposed to make the assignment of categories to a web service or components a much more straightforward and light-weight activity. In particular, the ability to assign multiple tags to web services is expected to increase the probability that users can find a suitable web services. Nevertheless, publishing web services or components based on the manual assignment of tags remains a hit and miss affair as recent experiments described in [17] underpin.

A comprehensive overview of component retrieval techniques is given by [9]. Other approaches that have been tried in the past include signature matching [13] or behavior sampling [18] and would in principle be applicable to web services, too, but they have either been too inaccurate or too time consuming to be acceptable for practical use. Furthermore, modern component-based development approaches such as KobrA [19] recommend that components be selected based on their specification, i.e. their interface and corresponding operation specifications. Our merobase search engine offers two algorithms that at least support the first part of this

[1] Merobase is our own code and service search engine.

postulation when searching for components – i.e., name based searches, which explicitly match the query string to the name of indexed components, and full abstraction-based searches which allow components to be found based on the elements and names in their interfaces. Our recent experiments described in more detail in [14] show that these name and interface-driven forms of search provide significantly better precision than simple text-based approaches. For example, the following table shows a comparison of the various component retrieval techniques. We performed twelve queries for functional abstractions such as –

```
isLeapYear(int):boolean
```

on the merobase repository (with almost 4 million Java source components available) and investigated the top 25 results using the four different retrieval techniques for their relevance (i.e. whether they delivered the expected functionality):

Table 2. Comparison of retrieval techniques on a large component repository

Retrieval Technique	Signature Matching	Keyword Searching	Name Matching	Abstraction Matching
Average Precision	1,1%	9,4%	14,9%	39,4%
Standard Deviation	2%	12%	15%	21%

These results indicate that abstraction matching is in fact the most precise query technique and hence more useful than the others for large repositories with millions of components. In fact these more sophisticated forms of abstraction search are essential in order to effectively find web service descriptions using Lucene-like indexing technology, since by definition web services are characterized (i.e. described) entirely by their interfaces and do not contain source code in the traditional sense. Finally, the algorithms can be fine tuned by giving additional weight to certain elements extracted from the query. For example, we are currently experimenting with improving keyword-based searching by taking elements such as the component's URL into account as Google does for regular web pages.

3. Web Service Indices

Another essential prerequisite for a usable web service search engine is an index of the currently available web services on the Internet (based for example on Lucene or some other similar indexing engine). Although the creation of source code and web service indices might at first sight seem rather similar, since they are both essentially textual descriptions of software components, in practice they present rather different challenges. This is because code search engines do not have to rely on crawling the open web to populate the index. Most of the software accessible over the Internet is contained in well known version management repositories (e.g. CVS or subversion) or is packaged in archive files such as tar, jar or zip files. It is thus possible to obtain a very sizeable source code index without doing any actual "crawling" in the traditional sense, and many of the main code search engines rely solely on these sources of software for their content.

Such sources are not available for web services, however, and hence, finding and validating suitable content is one of the biggest challenges involved in generating a web service repository. In the next subsection we discuss these challenges in more depth and describe how they can be addressed.

3.1. Index Creation

There are two basic ways in which a web service brokerage engine can populate its index of web services. One way is through the explicit publication efforts of web service developers and the other is by means of some kind of "crawling" activity which is focused on finding and analyzing web services. Virtually all attempts to set up web service search engines to date have been based on the first approach. As mentioned in the previous section, most of the public web service brokering services offer keyword-based search technology in which services are indexed by category. Crawling for web services presents some special challenges. Since WSDL files do not contain the additional metadata specified by UDDI, browsing by category will not be possible at all and hence such an index has to focus on the advanced retrieval techniques described above. Another challenge that is faced by all web crawling engines is the fact that there is a certain number of artifacts that can not be reached directly because they are not referenced in a publicly visible part of the web. In general, the only practical solution to this "hidden web" problem is to allow users to draw a search engine's attention to hidden "places" by inputting suitable links. This falls short of full scale web service publication in the UDDI sense, but is a useful complement to it.

Crawling the visible web for WSDL descriptions of web services presents two basic challenges –

- recognition of valid WSDL files and dependencies
- detection of properly working web services

Firstly, the recognition of WSDL files when crawling is hampered by the different file endings used for WSDL files. By convention, different web service environments typically use different file endings. While it is easy to recognize .NET and Java web services, crawling for WSDL files that end with the file extension ".xml" presents some difficulties. Since there is a vast number of XML files on the web, identifying those that are WSDL files requires significant effort. Basically, all XML files have to be processed and analyzed even though the proportion of XML files containing valid WSDL syntax is very low. The web service standards allow WSDL to be stored in any arbitrary XML document file, but for effective crawling this is a real obstacle that can only be overcome using large-scale crawling infrastructures that are able to process huge amounts of files. The actual parsing of WSDL syntax and the resolution of dependencies in WSDL files can be performed using open source web service frameworks like Apache Axis [15] that automatically download embedded WSDL fragments from external files.

Secondly, the current ratio of working web services to published WSDL files on the Internet is very low. To avoid the indexing of unavailable web service components it is necessary to check the on-line availability of the service. Therefore, it makes sense to test the availability of a potential web service by invoking one of its methods using randomly generated test data. This can be stored in a database and used for later periodic re-evaluation of the web service's availability. The receipt of any valid SOAP response can be interpreted as an indication that the service is at least responding and can be regarded as being "live".

3.2. Index Maintenance

Since web services are remotely executed components that are under the full control of the service provider, they can be removed from the web at any time, either on a temporary or on a permanent basis. Index maintenance is therefore very important to the perceived quality of a web service repository. Conventional search engines like Google and Yahoo keep their content up to date by recrawling the web on a periodic basis. However, for search engines that index web services this is not possible because the availability of a WSDL description does not necessary imply that the web service is on-line and working. Similarly, the disappearance of a WSDL description does not imply the disappearance of the web service itself since WSDL files can be distributed independently of the hosting web service environment. Most of the previous attempts to set up web service repositories did not check the validity of their contents at all and thus over time they contained more and more retired services that had been shut down. Without appropriate automated checking mechanisms, the only way to remove or mark retired services is to check them manually. However, this is impractically once the repository expands beyond a certain size. Therefore, the collection of web service status information - such as whether they are working properly, are under revision or have been retired – needs to be automated.

The only way for a search engine to deal with these issues is to periodically test the availability of the web services in its index, or to test their availability before delivering a search result and filter out those that have been retired. However, the latter approach can significantly lengthen the time required for result sets to be generated. Generally speaking, there are two basic ways of determining the state of an indexed web service. One way is using information supplied by users when they try to test a service through the included execution engine. The other way is to implement some kind of background "liveness" testing.

The merobase repository has a built-in execution engine which can detect the unavailability of a service. It is also able to collect data that was provided by users when using the execution engine to test a service. These user-provided data sets can be retrieved randomly from our database and used to support background liveness testing in a relatively straightforward way. When a service is identified as unavailable through a user-driven execution attempt or a background liveness check the service's index entry has to indicate that it is currently unavailable. At the same time, the periodic time interval for background availability testing has to be decreased to a shorter time interval. Using randomly generated test data the service can be placed under "observation" for a certain time until it is designated as a "retired-service-candidate". Once a further number of tests have failed the web service can then be finally removed from the index. Automatically keeping the index up-to-date in this way greatly reduces the number of "false results" that are returned to users and thus increases their perception of the search engine's performance.

4. Additional Challenges

As explained in section 2.2, simply searching for keywords leads to imprecise search results in large repositories. This is especially true for web services, as there is no source code available against which the keywords from the query could be matched. For web services it is therefore only practical to use the name-based or abstraction-based queries. And as outlined before, of these two the abstraction-based queries which find services based on their interfaces are the most precise. However, the definition of what the interface of a service is from the point of view of a service requestor (i.e. the user of a search engine) and a service provider may not always be the same.

For example, the user of a web service often does not care about additional "management" parameters such as session or user IDs which have to be provided in service operation invocations but are not directly related to providing the desired functionality. Session IDs are particularly problematic because by definition most web service clients are designed to hide session ID values from the user. However, the whole point of client/server technology is to provide the user with the illusion that he/she is the sole user of a service whereas in fact there are usually many concurrent users. The session ID is thus a critical parameter of the operations in the server interface, but is hidden from the human user of the service. Nevertheless it is a common pattern to include session ID parameters in method signatures. Thus, simply matching an interface query as defined from the perspective of the user to the actual interface supported by the web service would not lead to the desired results. Since WSDL only allows web service descriptions at a low level of abstraction, we have introduced the notion of the "pan-client" and "per-client" views of a service [1]. Another frequently used pattern we identified in this context is the usage of authentication data parameters, e.g. username and password or license keys, which either have to be sent once or with every method call.

As an example, consider the following web service which simulates the millionaire quiz. A user can start the quiz by calling the startNewGame() method. He then receives a session ID which has to be used to identify the game in subsequent method calls. Additionally, the user has the opportunity to start multiple games at any time by acquiring multiple session IDs. From the startNewGame() method call, he also receives the first question. The question can be answered by the continueGame() method. Two methods returning either a reduced set of remaining answers or answer probabilities (simulating the "fifty-fifty joker" and results from an audience survey) are also available. These methods can only be used once per session. Each method returns the session ID so that a service client, which could run multiple quizzes at the same time, can assign the return messages from the provider to the corresponding quiz. The following figure summarizes the service's actual – i.e. the pan-client – interface in a graphical form.

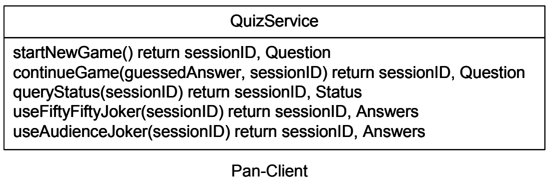
QuizService
startNewGame() return sessionID, Question
continueGame(guessedAnswer, sessionID) return sessionID, Question
queryStatus(sessionID) return sessionID, Status
useFiftyFiftyJoker(sessionID) return sessionID, Answers
useAudienceJoker(sessionID) return sessionID, Answers
Pan-Client

Figure 2. Pan-client view of a quiz web service

Although it is necessary to provide the session ID parameters when invoking these methods, it is unlikely that a user of a search engine is either interested in session IDs or even running multiple quizzes simultaneously. From a client's point of view, only questions and answers are important in the QuizService abstraction. Thus a search request for a QuizService is more likely to utilize the per-client perspective as shown in the following figure.

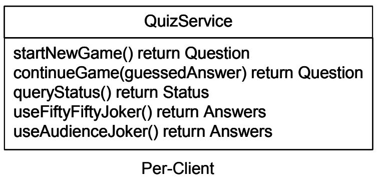
QuizService
startNewGame() return Question
continueGame(guessedAnswer) return Question
queryStatus() return Status
useFiftyFiftyJoker() return Answers
useAudienceJoker() return Answers
Per-Client

Figure 3. Per-client view of the quiz service

As figures 2 and 3 illustrate, the per-client view is much simpler and more concise, so we believe that searches should be done with the per-client specification of the service. Even if the service does not always return a session ID, the session ID always has to be transmitted as an input parameter. A user not interested or aware of sessions would query a component search engine without the additional parameter (e.g. startNewGame(): Answers) and not retrieve the desired result (QuizService). An advanced user aware of session IDs could also fail to get the desired results because the position and the type of the session ID may vary. Although the merobase search engine supports permutations of parameter orders, no component search engine we know of directly supports different parameter variants. This would indeed be necessary since session IDs could be of various number or string types.

Consequently, an ideal component search engine should either map the per client view to a pan-client view, or recognize the per-client view of a web service in the first place. However, as there exists no widespread standard for separating the per- and pan-client views, we currently propose to use some heuristics during crawling. A search algorithm could check if a return parameter of one method appears as an input parameter for other methods. Another clue could be the name of the method as well as the name and type of the parameter. Furthermore, methods for creating a session usually don't have any parameters besides authentication parameters (if at all). However, further research is necessary to evaluate the effectiveness of such methods.

5. Suggested Improvements to Web Service Standards

Using a combination of the crawling techniques outlined in section 3 merobase has been able to assemble a repository of about 3000 web services. Although this demonstrates that it is feasible to realize an effective web service repository, the task could be made easier by defining some conventions in the way that web service standards are applied and that web services are implemented.

The first convention we propose is that every web service should implement a standard “liveness” operation which can be used to check that it is still on line and has not been retired. This method would need no input parameters and would return a single standardized output parameter indicating its availability status. The uniform availability of such a simple method would greatly reduce the complexity involved in automating the creation of test data for arbitrary methods and greatly increase the ability of web service repositories to maintain the freshness of their contents.

In order to tackle the problem of finding the appropriate interface from a user's perspective, we propose some simple conventions for WSDL descriptions. In addition, to optimize the processing of existing WSDL descriptions by search engines, we advocate that they be automatically "marked" as WSDL documents. The goal of this convention is to identify parameters that are effectively hidden from end users and are only used to provide management information. For example, parameters used to identify individual users in a multi-user context should be marked as session ID parameters. Likewise, parameters needed for authenticating service users should be marked in a similar way. We propose to mark them by adding additional attributes (such as "sessionID" or "authData") to the XML schema data types definition in a WSDL document.

Since there will always be a large number of web services which are not annotated in this way, at least for some time to come, heuristics of the kind proposed in the previous section have to be used for identifying these management parameters and indexing them accordingly. With marked WSDL interfaces, the pan- and per-client interfaces could be explicitly distinguished and thus the user could choose the appropriate views. In addition, for annotated WSDL interfaces an unambiguous mapping can be defined from the pan-client view to the per-client view and vice versa.

6. Conclusion

If web services are to fulfill their true potential and revolutionize the way in which enterprise software applications are written and the way in which businesses deliver software functionality to one another, a practical and effective service brokerage solution needs to be developed. If not, web services will remain little more than a convenient middleware and wrapping technology, and the envisaged market of services will remain an elusive vision.

In this paper we have outlined the main issues that has lead virtually all previous attempts to set up public web service brokerage services to fail – the underlying reliance on the human management of repository content. We then outlined the ingredients of an alternative, practical approach which adapts the technology used in emerging “code search” engines to provide useful searches over a repository of web services. Using a combination of these techniques, our merobase search engine has been able to assemble a repository of about 3000 existing web services. These are integrated into an index of several million source code components and around ten thousand binary components that are searchable using name and interface-based queries as well as simple text-based queries. Searches can also be restricted to web services using the “type:service” constraint.

Although 3000 may not at first sound like a large number, it is 10 times greater than the number indexed by the UBR at the time of its closure. In fact, to our knowledge it is the largest searchable repository of web service currently available on the Internet. This has been assembled from web services that existed before the deployment of the search engine. Once its availability becomes more widely known we hope that the size of the repository will be increased by the explicit publishing of components.

Based on the insights gained during the development of the merobase web service repository we have been able to identify several ways in which the basic standards underpinning web services could be improved to support web service brokerage. Further research remains to be done on how to create intelligent heuristics that allow the per-client view of the user to be derived from traditional WSDL documents, and how the availability state of web services can be automatically and efficiently checked at run-time.

7. References

[1] C. Atkinson, D. Stoll, H. Acker, P. Dadam, M. Lauer, and M. Reichert: “Separating Per-client and Pan-client Views in Service Specification”, *Proc. of the Int'l Workshop on Service Oriented Software Engineering* (IW-SOSE), Shanghai, China, May 2006.

[2] A.W. Brown, G. Booch: “Reusing Open-Source Software and Practices: The Impact of Open-Source Software on Commercial Vendor*s”*, in C. Gacek (Editor): *LNCS 2319*, Springer, 2002.

[3] D. Fafchamps: "Organizational Factors and Reuse", *IEEE Software* Vol. 11, Iss. 5, 1994.

[4] W.B. Frakes, C.J. Fox: "Quality improvement using a software reuse failure modes model", *IEEE Transactions on Software Engineering*, Vol. 22, Iss. 4, 1996.

[5] O. Hummel, C. Atkinson: "Using the Web as a Reuse Repository", *Proceedings of the International Conference on Software Reuse (ICSR)*, Torino, Italy 2006.

[6] M. Lenz, H. Schmid, P.W. Wolf: "Software reuse through building blocks", W. Tracz (editor): *Software Reuse: Emerging Technology*, Computer Society Press 1987.

[7] The Apache Software Foundation: Apache Lucene, http://lucene.apache.org, visited Apr 2007.

[8] D. McIlroy: "Mass-Produced Software Components", *Proceedings of a conference sponsored by the NATO Science Committee*, Garmisch, Germany 1968.

[9] A. Mili, R. Mili, R. Mittermeir: "A Survey of Software Reuse Libraries", *Annals of Software Engineering* 5 (1998).

[10] M. Morisio, M. Ezran C. Tully: "Success and failure factors in software reuse", *IEEE Transactions on Software Engineering*, Vol. 28, Iss. 4, 2002.

[11] UDDI Specification v3.02, "Universal Description Discovery and Integration", http://www.oasis-open.org/committees/uddi-spec/doc/spec/v3/uddi-v3.0.2-20041019.htm, visited Apr. 2007.

[12] J. Poulin: "Populating Software Repositories: Incentives and Domain-Specific Software", *Journal of Systems and Software* 30 (1995) 3.

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

[14] O. Hummel, W. Janjic and C. Atkinson: "Evaluating the Efficiency of Retrieval Methods for Component Repositories", to appear in *Proceedings of the International Conference on Software Engineering and Knowledge Engineering (SEKE)*, Boston, 2007.

[15] The Apache Software Foundation: Axis, http://ws.apache.org/axis, visited Apr 2007.

[16] R. Seacord: "Software Engineering Component Repositories", *Proceedings of the International Workshop on Component-Based Software Engineering*, Los Angeles, USA, 1999.

[17] T. Vanderlei, F. Durão, A. Martins, V. Garcia, E. Almeida, S. Meira: "A Cooperative Classification Mechanism for Search and Retrieval of Software Components", *Proceedings of the ACM Symposium on Applied Computing (SAC)*, Information Retrieval Track, Seoul, Korea, 2007.

[18] A. Podgurski, L. Pierce: "Retrieving Reusable Software by Sampling Behavior", *ACM Transactions on Software Engineering and Methodology*, Vol. 2, Iss. 3, 1993.

[19] C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B. Paech, J. Wüst, J. Zettel: *Component-based Product Line Engineering with UML*, Addison Wesley, 2002.