Program Comprehension for Web Services

Nicolas Gold[1] and Keith Bennett[2]

[1]Department of Computation
UMIST
Manchester, M60 1QD, UK.
N.E.Gold@co.umist.ac.uk

[2]Department of Computer Science,
University of Durham,
Durham, DH1 3LE, UK.
Keith.Bennett@durham.ac.uk

Keywords: Program Comprehension, Web Services, Software Maintenance, Service-Based Software

Abstract

Web services provide programmatic interaction between organisations within large heterogeneous distributed systems. Using recent experiences of constructing and enhancing a data integration system for the health domain, based on web services, we draw conclusions about new problems for program comprehension. These derive from the fundamentally dynamic and distributed nature of the environment. We suggest several key research topics for program comprehension, arguing that these are crucial if software constructed from web services is to be supportable over a long period. Finally, we briefly summarise some wider conclusions about understanding web services at the application domain level.

1. Introduction

It is well established, through an extensive series of surveys, that software maintenance is the dominant cost in the overall lifecycle for much software, and of that maintenance, software engineers spend a very large proportion of the time in trying to understand existing software, either to fix bugs, or more significantly, to enhance functionality. This primary activity is termed program comprehension, and refers to the application of techniques and processes that facilitate the understanding of software [8]. One of the aims of much program comprehension work is to understand models of comprehension that link the program and application domains (e.g. [5, 10]) and provide tool support for this process (e.g. [9]).

Application software constructed using web services adds new attributes which need to be understood by the maintainer who is planning to modify that software. The software underlying each web service will itself undergo maintenance during its lifetime, and we shall assume that this will be undertaken by conventional comprehension methods. At a higher level of abstraction, software may be constructed by integration services that will use lower level or atomic services to provide functionality. It is this integration level (e.g. in the form of a broker) on which we concentrate in this paper. Of course, a web service itself may be composed of sub-services, so the approach can be applied recursively. The aim of the paper is to define new research problems that are posed by a web services application architecture.

Using recent experiences of constructing a prototype data integration system with web services, we identify and amplify issues that pose problems for program comprehension of web services based software. Building and experimenting with this system has provided us with the insights to consider the implications and consequences for program comprehension.

The paper is organized as follows. We begin by describing web services. We then provide a summary of the prototype system; more details can be found in [3, 12, 13], and further contextual information is available in [1, 2, 4], and at www.service-oriented.com. Having presented this background material, we identify and explore program comprehension issues for software constructed using web services. Finally, we summarise these issues into areas for future work.

2. Web Services

Shirky [11] suggests that web services (WS) address application program inter-operability. The most general form involves binding or composing complex programs together from components anywhere in the world. General inter-operability has been tried before, but with partial success, for example DCOM, Corba, and RMI.

With these systems, both the client and server have to load the system; with web services, the idea is to know nothing about the "other end" other than such information as can be communicated via standard protocols. Web Services Description Language (WSDL) allows the description of a web service so that a call for it can be assembled and invoked from elsewhere. In order to communicate data in a system-independent way, XML is used. A Universal Description, Discovery, and Identification (UDDI) registry (or registries) allows service vendors to publish and offer services, and users to locate and call them using WSDL descriptions published along with service identification information.

Software has previously been developed, delivered and maintained as a product. The internet is stimulating interest in software which is instead delivered and used on demand (in so-called "internet time"), because (for example) this potentially allows faster and more flexible evolution to meet changing business needs. With a service-based approach, there is a much looser coupling between business requirements and software solutions. However, actually implementing this is far more complex than technical considerations alone would suggest. For example, services need to be identified, selected, and procured. This may well require negotiation within a market [7]. The consumer application will need to have confidence that the service performs as described, and if not, that means of redress are available. Fundamentally, the service model will fail if there is a lack of trust between vendors and users. The overall research of the Pennine Research group of which we are members is therefore directly concerned with these wider problems [2].

To test the group's research findings and to scope the requirements for an integration layer in the web services stack, we have undertaken a prototype implementation of web services, addressing a real life problem. We chose, as the prototype application domain, the issue of integrated health care data. This was a highly appropriate case study because it combines the need for flexible software (functionality) with issues of independent, heterogeneous data sources (data) which similarly need the type of ultra-late binding required by the system's functionality. We wished to explore the extent to which these could be met by a web services solution, and our prototypes have enabled us to determine issues of program comprehension.

3. The Experimental System

3.1. Problem

Let us consider a case in an accident and emergency department in UK Health Care. Mr Smith collapsed when he travelled from Manchester to Durham and has been admitted in an unconscious state. He had previously lived in Leeds, Newcastle and London before moving to Manchester. Therefore, his treatment history data are stored in different systems at different sites using different technology. To treat him properly, the doctor in Emergency needs to know as much information as possible about Mr. Smith (for example, any drugs he is currently taking, so that drugs needed now can be chosen safely). He needs to acquire an integrated view of the data from autonomous, heterogonous and distributed data sources and these data sources may change. As one small example, there are 56 different types of mental health service alone in the UK; each geographical area will typically have several (or many) of such services [16].

We have completed the first prototype in the EPSRC-funded IBHIS (Integration Broker for Heterogeneous Information Sources) project. This project uses the UK's National Health Service as an example of a large complex service-based environment. Current web services technology has been used extensively and largely successfully in this distributed prototype system. In order to gain a good understanding and model of the domain, extensive activities have been undertaken involving health service professionals; these are not reported here.

3.2. Architecture

In order to provide the end-user (e.g. the doctor) with a unified view of data on demand from autonomous heterogeneous data sources, we use a *Service-Oriented Data Integration Architecture* (Figure 1). Further details can be found in [12, 15].

We assume that all the elements such as medical content (vocabulary), constants (postcode etc), datatypes and intent are defined or described with an ontology-based approach by the NHS authority within the NHS domain. This ensures that all data elements are semantically correct and can be interpreted correctly by service requesters and providers. Service providers (including data service providers) implement services and describe them using service description languages such as WSDL [17] and DAML-S [18]. All web services may be implemented using different languages on different platforms; they must of course use agreed protocols. Service providers publish the service description into a service registry, such as UDDI, and map the local data elements to the standard terms defined by NHS authority.

Services are then located using the service registry and the service implementation is invoked via SOAP. When an end user wants to acquire integrated data they look for the UDDI registry and finds a suitable Integration Broker Service (IBS) that satisfies the requirements. When the IBS is invoked, it dynamically finds the sub services (data

and functional services), and then binds to them appropriately.

The discovery service is used to discover and bind to service implementations at run time. When there is more than one service providing the same function, it can be used to choose one service based on the user's requirements.

The ontology service is used to provide semantic transformation for different data items. The security service is used to authenticate the user and control the data items a given user can access.

The Integration Broker Service (IBS) is a composed service, which integrates different data access services (DASs) and functional services such as the Discovery Service (DS), security service, and the ontology service in order to provide the end-user with an integrated uniform view of the data. The backbone is the workflow service, which manages the business logic, for example, how different data services work together to provide integrated data. It combines the business process and data integration. The IBS could also compose a negotiation service to perform negotiation with the available data sources.

A Data access service (DAS) is a variation on the typical (Web) service as it is more data intensive and is designed to expose data as a service. Data service providers implement the DAS, which may query multiple, heterogeneous data sources. Alternatively, different DASs may query the same data source but produce different data outputs. The data service providers in the NHS domain must describe the data input, output and data format, the methods to acquire the data, the security requirements to use this service, and data service version.

When data service providers publish the DAS description file into the registry, they publish the DAS metadata and ontology.

For change management, DAS providers may provide users with different versions of the DAS. The service consumer can decide which one better meets their requirements, or choose to bind to the latest version by default.

There is no integrated schema; data is integrated on the fly.

The initial prototype broker (Figure 2) was completed in the Summer of 2003, and a second prototype is now under way, to exploit the lessons learned. In particular, the first prototype provided a thorough test-bed of the technology (which was mainly IBM's websphere and Java J2EE) but it was essentially static. The second prototype is employing dynamic binding far more extensively.

3.3. Results from the Prototype

The following represent the main results from the prototype which are of particular relevance to program comprehension:

1. *Dynamic web services:* even in our (largely static) prototype, it is clear that a "program" may only be defined at run time, when it is executed As a simple example, we have found that a remote procedure call mechanism for service invocation is far too restrictive, and that a document style of call is essential. The timescales which are available to comprehend service based software may be very much reduced; changes may be demanded very quickly.
2. *Web service description:* the WSDL description of a service plays a crucial part in locating, binding, and understanding. Currently, WSDL is mainly aimed at end-point identification, to allow calling. Much more work is needed on extending WSDL for applications, for stateful services and for non-functional attributes. We found that ontology services will be essential in large organisations such as the health services (where there is no standardisation of terms); there is unlikely to be a single ontology (requiring mappings), and both single and multiple ontologies evolve over time.
3. *Security:* there are many new issues of privacy, security and access control (see for example [14]). In general, these are not discussed further here.

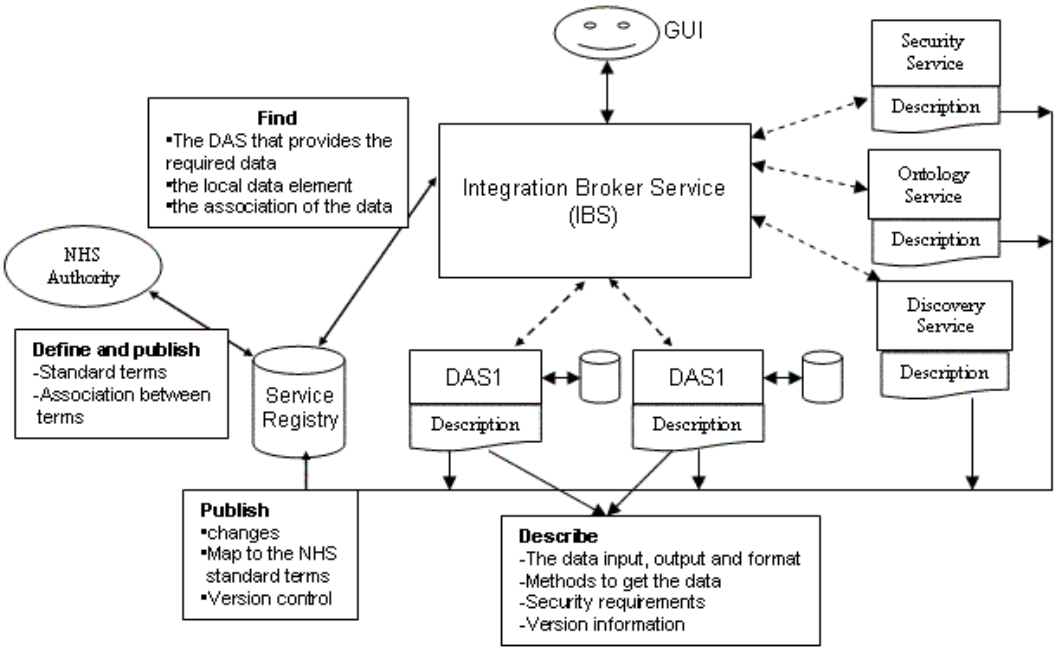

Figure 1: Service-Oriented Data Integration Architecture

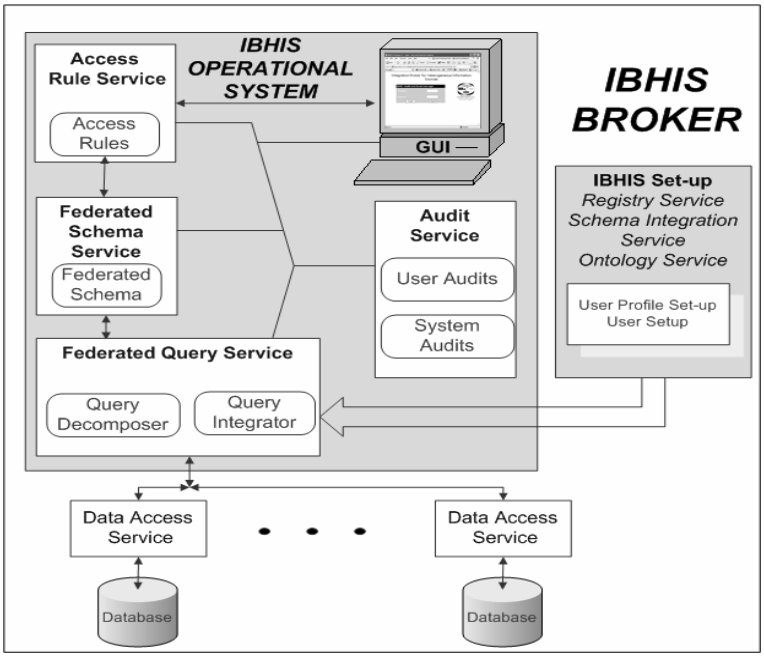

Figure 2: Conceptual Architecture of Prototype Version 1

4. Research Problems and Directions

In this section, we present a range of issues affecting software comprehension in the context of web services. We begin by examining the nature of the artefacts with which we are concerned i.e. the software itself. Our analysis then broadens to account for the environment in which this software operates. Finally, we examine the impact of these issues on existing approaches, identify some new requirements and lay out areas for future work. We approach our examination of the issues from the viewpoint of a software maintainer attempting to comprehend a program that uses web services. This approach is useful because in the event that we are maintaining a web service itself, the comprehension problem is really no different to that for traditional software. The novel problems arise when we are using web services rather than defining them.

4.1. The Problem Space

4.1.1 The Nature of Web-Service Based Software

Software constructed partly or wholly from web services has a somewhat different character to the traditional source code we are used to working with. Although many aspects are similar on the surface (e.g. meaningful names for subroutines to convey meaning about their purpose, remote procedure calls for distributing applications), deeper analysis shows how software that uses web services requires a different approach to understanding.

Rather than comprehending a file (or files) of source code where procedure calls carry as much meaning as can be fitted into the name, a WS-based application has access to a much richer description of the software embodied in the services it uses. Ideally, this information should be contained in the WSDL comprising the UDDI entry for the service and can thus provide a maintainer with a detailed description of that part of their program (although at present, UDDI does not have the descriptive power necessary to achieve the full potential of this approach, and an important role for the program comprehension community is to influence the development of web service descriptions). This is essentially documentation on a fine-grained level and may suffer from the usual problems of being out of date or inaccurate. However, in a service-oriented context, there is a greater motivation for it to be accurate and up to date because it is likely that service usage is being sold by the service developer and it is therefore in their interest both to attract and to retain clients through good quality information. The consequence of a fine-grained documentation-driven approach to comprehension is that the timescale of understanding should be reduced and consequently evolution of the software system should take place more quickly.

Program comprehension occurs in various parts of the maintenance lifecycle e.g. impact analysis, prior to making changes, and debugging. For those activities where comprehension is needed to determine *what* is taking place in a system (e.g. comprehension prior to changing), the timescales should be reduced (assuming a sufficient level of detail is available). In addition, a web services architecture may force a better separation of issues and encapsulation (because the web services market may force this). However, in situations where the maintainer needs to understand *why* something is happening (e.g. debugging), it may be that the timescales are increased because the lack of detailed information about the internal operation of the software.

Although the maintainer thus has a potentially rich source of information about the services used, detailed construction of a mental model of the services and their interaction may be hampered by the fact that dependency information cannot be easily exploited in the way that it might be normally i.e. a maintainer cannot "drill-down" into the code embodied by the services in order to fully (or at least, sufficiently) understand the behaviour of that code. The descriptions are *all* they have to work with. It is possible to envisage a situation where an entire application is constructed from calls to web services with minimal or no intermediate processing at all. This would leave a fairly simple program to understand at a high level since it would only contain sufficient data structures to transfer the state of the data being processed from one service call to another. In a sense, this is an attractive model for programming since the program could be constructed solely from descriptions of its components; the level of programming abstraction thus being raised substantially. The issue of whether such descriptions are trustworthy remains however. In addition, it is questionable whether it is possible to form the kind of mental model required to understand and reason about software for change, solely from high-level, abstract descriptions.

Web services move away from a remote procedure call (RPC) approach to distributed applications (although they can support this architecture) and towards document-style interaction between parts of the software. RPC tends to have fixed numbers of parameters with fixed types and is often language-dependent. This differs markedly from the loose, flexible approach of web services (so-called document style interface) where interfaces are defined by their WSDL descriptions which can be changed easily and without reference to the implementation platforms and languages at either end of a connection. This approach gives us greater visibility (and perhaps

description) of the interaction between services but less long-term and local control over the mechanism. In comprehension terms, this means that we have additional artefacts (the interface descriptions) to retrieve and understand. It has been claimed [6] that the delayed binding and dynamic nature of WSDL interfaces will allow calling programs to adjust for changes in the services they use and eliminate the need for revalidating the whole system after changes are made. We disagree with this view. As we are not in control of the interface, it could change at any point in the future and do so without reference to us. When we come to understand software built on web services, we are thus required to take extra steps in order to determine the current interface to be used (e.g. by re-reading the WSDL file describing the service end-point). Changes by the service supplier to the interface could render our software useless until it is modified to match the new calling parameters (contrary to the view expressed in [6]). Alternatively, we must build sufficient flexibility into our code to cope with minor interface changes (although it is not clear how this might be achieved). Either way, complexity has been added, either to the understanding process or the artefact itself.

Although this complexity is a disadvantage, the greater visibility afforded by the readability and traceability of SOAP messages may partly compensate. Since we have access to the interaction between services, we can build a picture of the data flow between service and caller in terms of the actual data transmitted. Even if we cannot gain information about the internal operation of a particular service, we may be able to infer sufficient detail of its operation (from the transformation of data it performs) to effect our change. Such visibility is only available in the event that SOAP messages between caller and callee are not encrypted. If steps are taken to secure the information in transmission then we will need a way to decrypt it quickly to understand the operation of the program concerned.

In all of the discussion above, it is assumed that web services can simply be integrated (and consequently replaced) without difficulty. However, it is not clear how one would comprehend or characterise the level of abstraction at which services are offered and thus how their compatibility might be assessed. When understanding WS-based software, the actual size and complexity of a software system might be obscured behind calls to web services. For example, if we are attempting to understand a program containing 25 calls to web services, it is hard to determine whether this corresponds to 25 lines of code or 25000000 lines of code (and thus determine the internal and external complexity that might go with such sizes of program). Also, one cannot determine whether calls are made by these services to other web services to form an extensive network of services operating together. One might attempt to guess the complexity based on the description of a service (and thus its corresponding level of abstraction) but this is not a basis for good software engineering. These issues also make the estimation of maintenance effort required difficult.

A summary of the issues raised in this section is shown in Table 1.

Table 1: Comparison of Traditional and Service-Based Software

Characteristic	**Traditional Systems**	**WS-Based Systems**
Program-Level Semantics	Strong	Weak
Coupling	Tight	Loose, dynamic
Extent of Analysis Available	Comprehensive	Partial
Amenability for Static Analysis	High	Low
Visibility of Component Interaction	Low	High
Domain Semantics	Embedded in Source Code	Embedded in Descriptions
Assessment of Complexity	Easier	Harder
Interface flexibility	Low	High
Interface dynamism	Static	Dynamic
Ontology	Not used	Key component
Interface definition	Static, with explicit parameters and types	Document style

4.1.2. Context of Web-Service Based Software

The open and distributed nature of software based on web services requires us to consider the environment in which such services can be obtained and used as this is likely to have a greater impact on the software than in more traditional systems.

We have already raised the need for developers to trust the description of services provided by those who operate them. The issue is somewhat more complex than trust because developers need also to understand that description. This implies that such descriptions need to be accurate, complete, precise and so on; they must reflect the traditional "ideal" qualities for requirements statements since service procurement in this context is effectively a developer matching their requirements to the descriptions of available services. This is not a trivial problem. In a global market there may be many natural languages used to describe services. Within these, the most frequently used approach to description is to employ an ontology in order that terms are commonly understood (for example, in our health domain, the term "child" has very many interpretations, depending on which organisation is involved). However, the marketplace for web services is likely to be so large that multiple ontologies (which themselves evolve) are likely to exist. There will thus be a translation problem to be solved to map terms from one ontology to another (assuming similar terms are available in both). This issue was highlighted by the IBHIS project which is planning to use the US-based SNOMED ontology [19].

The complexity of service procurement and management thus adds overhead to the comprehension burden (as well as substantially changing the artefacts to be comprehended). Technological standards may also dictate the structure and content of service descriptions and these may not be immediately compatible. Assuming developers understand and trust the intent of the descriptions provided, they must also trust that services will perform as described i.e. they need to know the typical operation of a particular service. This may carry financial implications in terms of the paying for service usage.

So, in addition to the purely functional and technological view, one must also consider the economic implications of web service use. Simply knowing what a service is supposed to do (functionally) and trusting that it will do it, is sufficient only if the service is provided by the developer's organisation. In most cases, we envisage the advantages of WS technology arising from the purchase of service usage from other organisations. This implies the existence of contracts between the organisations for service supply and consumption. Consequently, in understanding programs that adopt web services, one must be aware not only of the functional characteristics, but also of the contractual terms that govern their use. This may affect when a service can be invoked, the behaviour or performance of that service at particular times of day, or possibly the process to be followed in the event of service failure. These aspects further complicate the comprehension burden for the software maintainer. As an example, an IBHIS data access service must be rigorously secured against accidental or malicious damage. A contract of use will have to specify which users may access the service and under what conditions.

In all of the above discussion we have not considered in detail the relationship of data to the source code that operates on it. Data may pose a less complicated problem for service-based software (or at least, can be reduced to the same set of problems as described above). If data is stored "locally" i.e. it is the responsibility of the local program, then the understanding problem is no more complex than it is currently. If stored using a service, access to data may be undertaken in a similar manner to accessing an object or black-box component: the actual representation and data structure does not matter to the user as long as the interface is clear. Thus data access becomes a functional query service which is subject to the difficulties of description and trust described earlier.

A related issue is thus whether the web services we are using are stateful or not. The above discussion assumes stateless services (i.e. execution simply takes place and results are returned) but it is entirely possible that services may retain state between invocations. This adds another comprehension burden because it is now important that the software maintainer using such web services has the ability to interrogate and describe the state of the services at various points during execution. This requires that a greater number of interfaces are offered by a service to support these additional queries (and consequently the maintainer has a more complex environment within which to work). The IBHIS project uses both stateless and stateful services (the obvious example of the latter is a patient database). This raises additional problems of how stateful services should be described; in IBHIS, meta database solutions are being explored [15].

Thus far we have raised a number of issues that complicate program comprehension when web services are used. The style of software produced in a web services context is different and, although potentially easier to understand at the highest level, requires great trust to be placed in the providers of supporting services. The environment in which the system operates thus assumes greater importance and software developers will need to take greater account of both the technological and

economic aspects of the environment that relate to their software.

4.2. The Solution Space

The previous sections have raised a number of problem issues for software comprehension. In this part of the paper, we describe some specific solutions to parts of these problems and distil our ideas into areas for future research

In general terms, we perceive much existing comprehension research to be focussed (rightly) on the static analysis of systems. In a service-oriented world, dynamic aspects of system behaviour become much more important. Indeed, it might be argued that with the very late binding characteristics of service-based software, the actual execution of the software is the only firm representation of the software solution that exists. Therefore, we need to adapt and refocus our research on understanding both static aspects and dynamic behaviour together. In particular, we need to provide software engineers with a full set of behavioural information about the services and software on which they are working as early as possible in the understanding process.

Essentially, this is a manifestation of the trust problem described earlier. To increase developers' belief in service description and behaviour, one could envisage "meta-services" appearing in the marketplace whose role is to determine the typical behaviour of other services by repeated testing on ranges of values, or using pre and post conditions to infer the properties of a particular service. A developer wishing to understand or verify the behaviour of a service in relation to its description could query one of these meta-services (at a cost) for the information. Since repeated execution of a service may incur repeated charges for its use, testing (either for understanding or regression testing) becomes an expensive activity.

The meta-service would gain return on its investment in testing costs by selling the information to many developers who wish to use it. The developers can gain the specific information they require at a fraction of the cost of determining it for themselves. The implication of such an approach to service information delivery is that the tools and environments we develop to support program comprehension will themselves become service-based: since we cannot afford to test every service ourselves, we must incorporate the information provided by someone else about services of interest into our tool and process environment. As the software we produce becomes more reliant on a marketplace of services to operate, the tools with which we create and maintain that software also need marketplace support to supply the information required for their operation. This does not preclude developers operating t

control mechanisms to protect confidentiality of data in the repositories. Another major issue in this area will be to identify what information is truly important to capture and what can safely be ignored.

3. *Architectural style:* Although it is early days, it may well be that certain architectural styles emerge as particularly favourable for web services applications. It is possible too, that such styles can be represented in some form of pattern. Standard (or typical) ways of implementing services can be of benefit to program comprehension, but this research will benefit from active participation by the comprehension community. The use of document-style interfaces may also benefit from standard practice.
4. *Trust management:* The next major research theme will be in managing trust in service descriptions. Describing services accurately and completely is very much an open research problem. Even if the description problem can be solved, it is still important that users trust the services they are procuring and judge them to provide value for money. This is a relatively new problem brought about by the automated aspects of these problems (resolving descriptive misunderstandings between humans is generally simpler than doing so by machine). There is a need to develop mechanisms in the marketplace and for individual maintainers to support those trying to understand and create software.
5. *Recursive web services:* We have noted that web services can very easily call sub-services, and so on recursively. This forms a supply chain and value chain structure in the software. Such chains will need research to understand their formation and evolution in the context of software services.
6. *Stateful web services*: We have noted that stateful web services and data intensive services pose new problems compared with programs (or databases) which retain state between uses. In particular, there are issues of description, access and security which relate directly to comprehension.

6. Conclusions

In this paper we have described comprehension issues for service-oriented software. Our views have been strongly influenced and informed by the experiences of building a prototype application, using web services, in the health service domain. We have explored the problem and solution spaces for programs that use web services and have extracted six major themes for future research. These address questions of how to collect comprehension information (including a representation of the dynamic behaviour reflecting late binding), to provide sufficient trust in the information such that it can be used, and how to analyse the information gained for comprehension. More generally, web services promise a major change in the way software systems are implemented, and marketed, and it is certain that maintenance and support will turn out to be very important issues in the longevity of such software. It is thus important that the program comprehension community engages with web services and software created with them, and influences the direction of technical and business applications.

Acknowledgements

The authors would like to thank other members of the Pennine Research Group for their contributions which have led to this paper. This includes: Paul Layzell, John Keane, David Budgen, Pearl Brereton, Jie Xu, Michael Rigby, Fujun Zhu, Mark Turner, Ioannis Kotsiopoulos, Michelle Russell, Michael Rigby. This work is partly supported by the EPSRC-funded IBHIS and CoMoS projects.

References

[1] **K.H. Bennett, N.E. Gold, P.J. Layzell, F. Zhu, O.P. Brereton, D. Budgen, J. Keane, I. Kotsiopoulos, M. Turner, J. Xu, O. Almilaji, J.C. Chen, A. Owrak,** A Broker Architecture for Integrating Data using a Web Services Environment, *Proceedings of First International Conference on Service-Oriented Computing (IC-SOC) 2003*, Trento, Italy, December 15 - 18, 2003.

[2] **K.H. Bennett, P.J Layzell, D. Budgen, O.P. Brereton, L. Macaulay, M. Munro,** Service-Based Software: The Future for Flexible Software, *IEEE APSEC2000, The Asia-Pacific Software Engineering Conference,* Singapore, 5-8 December 2000.

[3] **K.H. Bennett, J. Xu, N.E. Gold, M. Munro, Z. Hong, P.J. Layzell, D. Budgen, O.P. Brereton,** An Architectural Model for Service-Based Flexible Software, *Proceedings of 25th IEEE Computer Software and Applications Conference (COMPSAC)*, 8-12 October 2001, Chicago, USA, pp. 137-142.

[4] **O.P. Brereton,** The Software Customer/Supplier Relationship, *Comm. ACM,* Vol. 47, No. 2, pp 77-81.

[5] **R. Brooks,** "Towards a Theory of the Comprehension of Computer Programs", *International Journal of Man-Machine Studies*, Vol. 18, 1983, pp. 543-554.

[6] **E. Castro-Leon**, The Web Within the Web, *IEEE Spectrum*, February 2004.

[7] **A. Elfatatry, P.J. Layzell**, Creating a Mass Market for Software Services, *Comm. ACM,* to appear.

[8] **N.E. Gold, A. Mohan, C. Knight, M. Munro**, "Understanding Service-Oriented Software, *IEEE Software*, to appear.

[9] **N.E. Gold, K.H. Bennett**, "Hypothesis-Based Concept Assignment in Software Maintenance", *IEE Proceedings – Software*, Vol. 149, No. 4, August 2002, pp 103-110.

[10] **A. von Mayrhauser, A.M. Vans**, "Program Comprehension During Software Maintenance and Evolution", *IEEE Computer,* Vol. 28, No. 8, August 1995, pp. 44-55.

[11] **C. Shirky,** Web services and context horizons. *IEEE Computer*, September, Vol.35, No. 9 (2002) 98 – 100.

[12] **M. Turner, F. Zhu, I. Kotsiopoulos, M. Russell, D. Budgen, K.H. Bennett, O.P. Brereton, J. Keane, P.J. Layzell, M. Rigby,** "Using Web Services to create an Information Broker", *Int. Conference on Software Engineering, Edinburgh, 2004,* to appear.

[13] **M. Turner, D. Budgen, O.P. Brereton**, Turning Software into a Service, *IEEE Computer,* Vol. 36, No. 10, October 2003, pp 38-44.

[14] **E.Y. Yang, J.Xu and K.H.Bennett**, A Fault-tolerant Approach to Secure Information Retrieval, *Proceedings 21st IEEE International Symposium on Reliable Distributed Systems*, Osaka, October 2002.

[15] **F. Zhu, M. Turner, I. Kotsiopoulos, M. Russell, D. Budgen, K.H. Bennett, O.P. Brereton, J. Keane, P. Layzell and M. Rigby,** Dynamic Data Integration using Web Services, *submitted to the Int Conference on Web Services, San Diego, 2004*.

[16] http://www.dur.ac.uk/service.mapping/amh/

[17] http://www.w3.org/TR/wsdl

[18] http://www.daml.org/services/

[19] http://www.snomed.org/