Understanding Interoperable Systems: Challenges for the Maintenance of SOA Applications

Laura White, Norman Wilde, Thomas Reichherzer, Eman El-Sheikh, George Goehring
University of West Florida
Pensacola, Florida, U. S. A.
{lwhite, nwilde, treichherzer, eelsheikh}@uwf.edu,
gng@students.uwf.edu

Arthur Baskin, Ben Hartmann, Mircea Manea
Intelligent Information Technologies
Indianapolis, Indiana, U. S. A.
{abaskin, bhartmann, mmanea}@intelligent-it.com

Abstract

Software interoperability is a pressing need to allow governments and businesses to function efficiently. The most commonly recommended technology for interoperability is Services Oriented Architecture (SOA) implemented using web services. Several authors have argued that SOA systems may be particularly challenging to maintain, largely due to difficulties in program comprehension. Program comprehension for SOA could be aided by appropriate software tools to provide information to SOA maintainers. However, there is little experience regarding the questions that SOA maintainers will need to ask. This paper describes use of a prototype SOA search tool in an informal requirements elicitation study to gather feedback from practicing programmers about what SOA maintainers will want to know. Several specific information needs were identified, including the need for a compact way of representing datatypes used in services, and the need for ontology support to help understand the many different elements and attributes in web services descriptions.

1. Introduction

In both business and government, interoperability has become an indispensable requirement for information systems. Interoperability broadly refers to the ability of different systems to work together, especially by exchanging data and functionality during run time [1]. Private businesses increasingly operate as partnerships spanning different companies, often on different continents. Products offered by such partnerships need software support to track production, inventory, customers, etc. Each partner brings its existing software systems to the partnership so the different systems need to interoperate for the partnership to succeed.

Similar problems arise in the government domain. Traditionally many government systems were developed as "silos" each holding its own data and performing its own mandated functions. As such systems proliferated opportunities for synergies were lost, so the inevitable overlaps and workarounds became increasingly costly. Governments have realized that they can no longer afford silo systems. For example the US Department of Defense has mandated that interoperability shall be a criterion for all new software acquisitions [2]. In general, the vision of electronic Government with agile, effective and citizen-centric operations, clearly requires integration of computer systems across organizational boundaries. Service interoperability is a fundamental requirement for achieving these goals [1] [3].

However in this paper we argue that, as interoperable systems mature, they will present new and costly challenges for software maintenance. We focus on Services Oriented Architecture (SOA), one of the most common ways of achieving interoperability, and briefly discuss the very limited literature on SOA maintenance.

We then describe exploratory research on software tools to address these challenges of interoperability. Specifically we describe SOAMiner, a prototype search tool for SOA maintainers. SOAMiner is not a finished tool, but rather a research aid to explore what maintainers will need to know in the SOA maintenance environment. We describe an informal requirements elicitation study in which SOAMiner was applied to PAVERTM, a large services-based Civil Engineering system used by the US Department of Defense and other government agencies1. Based on this experience we give our perspective on the future needs for SOA maintenance and how program comprehension tools might help

---

1 PAVER is a trademark of the US Army Corps of Engineers
control the maintenance costs of interoperable systems.

2. SOA and Software Maintenance

Definitions of SOA vary [4]. However most of them describe composite applications constructed by orchestrating loosely coupled services running on different nodes and communicating via message passing. Typically the nodes and the services have different organizational owners. Often an infrastructure layer, sometimes called an enterprise service bus (ESB), mediates service interactions providing functions such as message routing, reliable messaging and data transformations.

There are many ways to implement SOA, but the most often recommended platform is the web services set of standards which help facilitate interoperability [4]. These standards include Web Services Description Language (WSDL) for service interfaces, XML Schema Definition (XSD) for service data types, SOAP for message formatting, HTTP for message exchange, and Business Process Execution Language (BPEL) for service orchestration code.

Some concern has been expressed in the literature as to the maintainability of large SOA composite applications. In Software Engineering, the term "maintenance" traditionally refers to all work done after the initial deployment of a software system. Such work tends to become progressively more expensive as a system ages. One reason is that, as pointed out by Belady and Lehman in their classic 1976 paper, system entropy (i.e., unstructuredness) grows as changes are made [5], making it more difficult to introduce further changes without also introducing new faults. A second reason is simply that development personnel move on to other responsibilities and thus knowledge of important details of the design is lost. The system slides gradually into a state called servicing, in which only limited changes may be attempted because too much knowledge has been lost. Since the system can no longer adapt, its value to the business is much reduced [6].

Thus sustaining program comprehension is a key to sustaining the business value of software. The first question a software maintainer must always ask is "How does this system work now?" It is very dangerous to make changes without deep knowledge of the pre-existing application.

Several authors have discussed special comprehension factors related to SOA that may make maintenance even more difficult than with earlier systems [7][8][9][10]. These factors include:

- The heterogeneity of SOA applications, so that maintainers may need expertise in many different languages and environments.
- The distributed ownership of services, so that for business reasons source code or key documents may not be made available to the maintainers.
- Poorly coordinated changes, as the different service owners are driven by different business needs, leading to multiple fielded versions of each service.

Several authors have looked at the organizational structures needed to support SOA maintenance. Kajko-Mattson, Lewis and Smith discuss the impact on traditional IT roles when organizations add SOA applications to their maintenance workload [11]. Several new roles emerge which need to be filled, and new problems arise, such as the prioritization of changes requested by different partners.

In a recent paper, Papazoglou, Andrikopoulos and Benbernou discuss the problems of managing versions as a SOA system evolves [12]. They distinguish between "shallow" service changes, which are localized in their impact, and "deep" changes which may cascade to other services. For deep changes they suggest that a gap-analysis model needs to be constructed, which would identify the differences between the "as-is" current deployed system, and the "to-be" system with the desired changes implemented. This proposal would seem to highlight the importance of program comprehension, since the creation of such a gap-analysis model would require deep understanding of the services making up the "as-is" system.

We are aware of only one paper that reports on experiences with program comprehension for a specific interoperable system. Gold and Bennett describe a research prototype web services system for integrating health care data from multiple providers [13]. The project used the UK's National Health Service as an example. The authors point out that there are many consequences from the widespread distributed ownership of services. Maintainers will generally have to rely on WSDL's as descriptions of external services, and WSDL's have many limitations for comprehension since they are designed primarily to allow runtime calling of the service. The maintainer will not be able to "drill down" into the code to get richer information since the code for the external service will probably not be available. They further argue that, since there are many service owners, the maintainer of a SOA composite application may need to deal with frequent unanticipated changes in the external services with which he is interoperating. He will need to re-read
the updated WSDL interface and analyze the changes to identify any necessary modifications to his own code. There may be a very short timescale to comprehend the modified service and make adjustments.

3. The SOA Maintenance Environment

We have argued that, while interoperability is now an imperative for information systems, providing interoperability may lead to costly challenges for system maintenance. One way to address these challenges would be to develop software tools to aid Software Engineers in understanding interoperable SOA systems.

We hypothesize a software maintenance environment similar to Figure 1. The Software Engineer is working on a SOA composite application that includes several "in-house" services (shown in blue) as well as several clusters of external services owned by other agencies. For the in-house services full information is available including source code, development documentation, and possibly even some support from knowledgeable colleagues. For the external services the situation is closer to that described by Gold and Bennett; the main information sources are WSDL's and XSD's.

WSDL’s and XSD's are not easy documents for a human to navigate. They have XML structure, which means that they consist of large numbers of interrelated XML tags. Just as one example, suppose a maintainer is trying to track down the data being sent to an external service. A WSDL 1.1 service interface description will have a <service> tag that refers to <binding> and <portType> tags. These two tags contain matching <operation> tags that specify, on the one hand the service messaging style, and on the other hand the input and output message content by referring to <message> tags. These <message> tags in turn refer to type declarations which may be in the same WSDL file or elsewhere, and so on and so forth [4].

These documents may also be quite large. For the PAVER system that we will describe in a subsequent section the collection of WSDL’s consists of ten files with a total of 49,000 lines while there are 63 referenced XSD files with a total of 14,000 lines.

Finally, the flexibility of the web services standards means that different developers using different tools may have produced quite different structures. Just as one example, message data types may be defined either directly within a <message> tag, or at the beginning of the WSDL in the <types> section, or in a separate XSD file. Maintainers will have to deal with the different styles adopted by each developer and so may have great difficulty knowing where to look for key information.

4. Tool Support for SOA Comprehension

There has not been much research on maintenance tools for SOA. A few groups have proposed dynamic analysis approaches to aid in SOA understanding [14] [15] [16]. Dynamic analysis involves collecting execution data, such as a trace, from a running system in either a test environment or in a live, deployed environment. Analysis of execution data can be a powerful approach, but is not always practical when dealing with large systems running across many nodes.

We are investigating a simpler static approach, based on text mining and text search. Search techniques would seem to be an attractive way to support understanding of heterogeneous SOA applications. Text search can be applied to many different file formats. There are open-source search toolkits available with proven efficiency. Perhaps most important, Software Engineers are familiar with performing searches so there is no need for them to learn a new kind of task.

However because of the newness of the field, we do not have a clear picture of what kinds of queries SOA maintainers will want to make, nor of what kinds of responses they will need. We have developed a search tool called SOAMiner as a requirements elicitation prototype that can be tried in SOA maintenance scenarios by Software Engineers with varying levels of SOA experience. The goal is to stimulate a conversation about "what SOA maintainers will really want to know" about a large, mature, composite application. While SOAMiner is currently a research prototype, our hope is that the answers to this question will allow it, or similar specialized search tools, to evolve into practical resources for SOA software engineers.
We envision a scenario in which a software engineer needs to make changes to an existing SOA application. The changes may be required because of some change within the organization, such as a newly required functionality or a software upgrade. Alternately, the in-house parts of the application may be stable but there are external changes such as the disappearance or modification of an external service. In either case the software engineer needs to determine how the application works now to devise a plan for handling the change.

Such changes often need to be made very quickly. Our experience working with software engineers maintaining earlier generations of software was that they were very busy people commonly working against tight deadlines. They were impatient with tools that were difficult to install or that required long practice to use effectively. If anything, the perpetual-beta status of SOA systems [10] will worsen the time pressures when it comes to SOA maintenance. In this environment, locating relevant system information quickly will be essential for cost effective maintenance. A basic SOA search tool requirement is thus speed; speed in tool installation, speed in document indexing, and speedy response to text queries. We have reported elsewhere studies showing that an early version of SOAMiner is very quick in indexing and querying [17], and we are currently working on speeding up the tool installation process. Our goal is that a SOA software engineer should be able to get useful search results about a large SOA application within one hour of downloading the tool.

After installing SOAMiner, the software engineer would point it at the different sources of information about the SOA application. These would certainly include the WSDL's and XSD's of external services but could also include folder hierarchies containing documentation, source code, configuration files, design notes, and so on. SOAMiner would build an index for these sources so that the software engineer could then search to answer questions such as "which services are dealing with tax data?" or "where exactly do we call the external addressLookup service?" Like any search web site, the tool could easily respond with a list of documents containing strings such as "tax data" or "addressLookup". However the challenge is to exploit the unusual structure of SOA documents to return the important matches with enough context to provide the software engineer what he needs to know.

We observed that many SOA artifacts have an XML structure; this is true, for example, for BPEL, WSDL and XSD files among others. Accordingly, our current version of SOAMiner focuses on indexing and searching the text contained within XML tags. SOAMiner is built on the Apache Solr search toolkit [18] with an interface derived from Ajax-Solr [19]. To use SOAMiner, first the parser is used to index any collection of files having XML structure. Then the collection can be queried using the interface shown in Figure 2. In this interface, the current query is shown on the upper left and the current set of result tags on the right.

SOAMiner uses faceted search, allowing users to start with a result set containing all of the indexed tags and then to add restrictions, either by entering text to be matched in the tag, or by selecting facets to choose a particular file type, file name, or tag name. In Figure 2, the user has first searched for <wsdl:message> tags and then drilled down to restrict those containing the text string "Inventory". Thirty matching tags have been retrieved and are shown on the right.

5. Case Study of the PAVER System

For the last year we have been conducting studies with successive versions of SOAMiner, first with students, then with faculty who have had more exposure to SOA. The case study with PAVER has been the first opportunity to try the tool using a full-scale industrial system and to work with programmers who are experienced in services computing.
PAVER is a pavement management system for condition-based maintenance management of airport pavements and roadways (See Figure 3). The version used in this study has 10 services with a total of nearly 400,000 lines of Visual Basic code.

Pavement engineers can use the system to: 1) develop an inventory of the pavements to be managed, 2) collect distress data about inventory items using an international standard for such field observations, 3) model pavement condition over time, and 4) plan budgets for repair work in order to make the best use of limited resources.

The PAVER system was originally developed in the late 1970s and has evolved over the years as both computer technology and best practices for pavement management have changed. Indeed, the use of the system by a growing number of pavement engineers in a number of countries has produced changes in the best practices, which have, in turn, required changes to PAVER.

The development of this system faces the twin requirements of 1) being able to utilize field data across a thirty-year period, and 2) make substantial evolutionary changes as computing and pavement management techniques change. In order to meet these twin requirements, PAVER has evolved into a closely coupled federation of modestly separate areas of pavement management functionality (e.g., inventory, condition modeling, and planning). Within the pavement engineering domain, each of these major areas can evolve somewhat independently.

PAVER is, however, not very typical as a SOA application because it uses SOA technology to manage the separation of what the user sees as a unified system into a family of closely cooperating but modestly independent components. The system uses Microsoft’s Windows Communication Foundation (WCF) to implement a family of cooperating services spanning the major modules within PAVER. As is common for WCF applications, the WSDL and XSD files, which can be used to describe the services in a standard way, are generated automatically from interface definitions using specialized WCF tags.

Figure 3. The PAVER Civil Engineering Tool

Interestingly enough, while at the moment, PAVER is largely used by itself, it is expected to become a part of SOA composite applications owned by others in the near future. Users who deal with "real property" of pavement assets need to integrate financial information with the Civil Engineering data managed by PAVER, so their view of PAVER will be similar to our hypothesized maintenance environment as shown in Figure 1.

The case study of PAVER was fairly informal, since the objective was to stimulate feedback on SOA program comprehension, not just to evaluate a tool. The WSDL's and XSD's were first accessed from the WCF server and then cleaned up to eliminate duplicate files and improve readability. They were then parsed and pre-loaded into SOAMiner on two laptop computers. To "prime the pump" for discussions, a set of questions was developed, loosely based on the kinds of search questions that Sim, Clarke and Holt found that Software Engineers used when maintaining traditional software systems [20]. These questions included both "concept location" queries that look for domain concepts (e.g., "What services / operations / messages deal with pavement inspections?") and "impact analysis" queries (e.g., "If I change this datatype, what code is affected?"). Finally each participant first did a small "warm-up" study using a very small WCF system so as to gain familiarity with SOAMiner and with the relation
between WCF WSDLs/XSDs and the underlying Visual Basic code.

The actual study took place in three phases; in each case participants were asked to imagine a scenario such as Figure 1 in which they were maintaining a composite application that accessed PAVER services. The phases were:

1. A graduate student with some SOA background went through the questions in a university setting.
2. A software engineer at the company that develops PAVER worked on the questions and related the tool's responses to his insight from developing the PAVER code.
3. Two software engineers from the PAVER development company tried SOAMiner and compared it with the information obtainable from Microsoft's Visual Studio™ programming environment. They used both the Visual Studio solution used in developing PAVER and a Visual Studio solution generated by asking Visual Studio to generate a stub client for one of the PAVER services.

In a debriefing and open discussion, the participants were asked for feedback on both conceptual issues (e.g., "What do you think a maintainer would need in this scenario?") and SOAMiner specific issues (e.g., "How could usability of the tool be improved?").

6. Case Study Results

The faceted search approach used by SOAMiner worked quite well for all of the participants although some confusing details of the user interface were noted for future tool improvement. All participants were proficient within about 30 minutes of first use. It seemed to them natural to start with a complete list of all possible XML tags and then narrow down the result by selecting facets or matching text.

It was interesting to see how participants could use their domain knowledge and exploit the search tool to get insight into the PAVER services. For example one participant wanted to identify which of the services acted as "owner" of a particular kind of data object. He used queries with programmer terms "create", "update", "delete" since he knew that these would typically appear only in the "owner" service and thus was able to quickly locate what he wanted. Search lends itself to this kind of man-machine partnership in which human knowledge compensates for deficiencies in the computer's vision of the problem.

Several participants noted that, while searches for services and for operations went fairly smoothly, searches related to datatypes were substantially more difficult. The problem is that the representation of a datatype in XSD's is very verbose, with the information spread across multiple tags, sometimes across multiple files. Since a SOAMiner search result shows individual matching tags, the user has to make multiple searches to figure out the whole picture.

Figure 4 shows one of the simpler examples. WCF has generated five tags encapsulating an empty <sequence>, simply to convey the information that the response to a ReportDelete operation is void.

```xml
<xs:element
   name="ReportDeleteResponse">
   <xs:complexType>
     <xs:sequence/>
   </xs:complexType>
</xs:element>
```

**Figure 4. A datatype description**

It is clear that SOA maintainers will want some more compact representation of datatypes, removing unneeded syntactic "sugar". We should also look for ways of identifying and summarizing common tool-generated patterns such as the one shown.

All of the participants thought that one good way to summarize WSDL and XSD results would be an expandable tree display, similar to Figure 5. The result of a query could be a pointer into a specific line within the tree structure. The user could expand or close the surrounding hierarchy levels to view context or exclude unneeded detail.

```xml
+ service
  + operation
    + input/output messages
      + datatype
```

**Figure 5. Expandable output tree**

A significant problem was observed with the WCF generated WSDL's and XSD's. To provide flexibility within PAVER itself, many of the operation signatures use interface types instead of concrete types. Abstract interface types contain only data definitions and method signatures, but contain no specific concrete implementation. They aid maintainability since they provide the possibility of using different concrete classes in the future while still preserving the interface. However for interface types WCF generates XSD's containing
used to address this need. However we have argued that maintenance of SOA systems will be costly and slow unless we lay the groundwork now for this future task. If governments are to use interoperable software to meet the needs of their citizens, then the problems of maintenance need to be addressed sooner, not later.

The first question a maintainer must always ask is "how does the software work now?" Yet there is very little research into program comprehension for SOA and in section 2 we have mentioned several reasons to expect comprehension to be unusually difficult.

As experience with SOA applications increases, we will gain more insight into the kinds of questions that SOA maintainers will need to ask. The conclusion of this study is that SOA-specialized text search looks like a good avenue to explore, but that maintainers need immediate additional assistance in understanding an application's data types and in removing syntactic "sugar" that obscures the meaning of XSD's and WSDL's.

There is a great need for additional practical research into search support for SOA. For example participants in this study suggested that ontology-based support could clarify the web services standards, their many extensions, and the domain terms found in SOA documents. Researchers need to find economical ways of building and maintaining the necessary ontologies, as well as ways of integrating them into search tools.

In the longer run emerging directions in search technology could have a major impact on SOA comprehension. In a recent widely-reported paper Etzioni has suggested that research into search needs to be more ambitious, and move beyond improving query matches to actually answering a user's questions [21]. He suggests that natural language processing, applied on a wide scale, could identify assertions about the world that underlie deeper understanding and thus provide correct answers. It would be very interesting to see how such methods could be applied to the SOA domain, where natural language text documentation lives alongside highly structured XML descriptions.

Software maintenance will undoubtedly continue to be the most expensive phase of software development and software comprehension will continue to be its major cost driver. Specialized search tools and techniques can help control these costs, but only if they are based on the real information needs and cognitive styles of SOA maintainers. We hope that this study has provided useful initial results as to "what SOA maintainers will want to know" and that these initial insights can be used by the software industry as it creates tools to support the interoperable systems of tomorrow.

2 We would like to thank Elliott Torres of the Blue Cross Blue Shield Association for suggesting the potential usefulness of domain ontologies for tools such as SOAMiner.
Acknowledgements

Work described in this paper was partially supported by the University of West Florida Foundation under the Nystul Eminent Scholar Endowment. We would like to thank UWF graduate students Douglas Leal, Joshua Dault and Juan Gil Restrepo, who implemented and tested different versions of SOAMiner.

References


