Journal of Software Engineering and Applications, 2013, 6, 435-445
http://dx.doi.org/10.4236/jsea.2013.69054 Published Online September 2013 (http://www.scirp.org/journal/jsea)

Towards Enhanced Program Comprehension for Service Oriented Architecture (SOA) Systems

Eman El-Sheikh[1], Thomas Reichherzer[1], Laura White[1], Norman Wilde[1], John Coffey[1], Sikha Bagui[1], George Goehring[1], Arthur Baskin[2]

[1]Department of Computer Science, University of West Florida, Pensacola, USA; [2]Intelligent Information Technologies Corporation, Indianapolis, USA.
Email: eelsheikh@uwf.edu, treichherzer@uwf.edu, lwhite@uwf.edu, nwilde@uwf.edu, jcoffey@uwf.edu, bagui@uwf.edu, gng3@students.uwf.edu, abaskin@intelligent-it.com

Received July 2nd, 2013; revised August 1st, 2013; accepted August 10th, 2013

ABSTRACT

Service Oriented Architecture (SOA) is an emerging paradigm for orchestrating software components to build new composite applications that enable businesses, government agencies and other organizations to collaborate across institutional boundaries. SOA offers new languages and a variety of software development tools that enable software engineers to configure software as services and to interconnect services with other services independent of differences in operating platform and programming and communicating languages. However, SOA composite applications introduce additional complexity into the construction, deployment and maintenance of software, for the purpose of aggravating the issue of program comprehension, which is at the heart of software maintenance. This article describes the challenges in SOA program comprehension and reports on the results of a two-part case study aimed at identifying information that would help a SOA software maintainer. Analysis of the results indicates a need for higher-level abstractions and visualizations that can enhance conventional text-based search to support SOA program understanding. This paper then reports on several specific abstractions, visualization methods, and the development of an intelligent search tool to enhance comprehension of the relationships and data within a SOA composite application.

Keywords: Service Oriented Architecture; Software Maintenance; Program Comprehension; Intelligent Search; Abstraction; Visualization; Web Services

1. Introduction

The emergence of Service Oriented Architecture (SOA) has created a new generation of information technology systems that support mission critical tasks in government and industry. Unfortunately, like each previous generation of software, SOA is also likely to create some new challenges for software maintainers.

Definitions of SOA vary [1], but generally they describe large systems-of-systems [2] in which composite applications are created by orchestrating loosely-coupled *service* components that run on different nodes and that communicate via message passing. An infrastructure layer, sometimes called an Enterprise Service Bus (ESB), mediates the communication, providing features such as routing, security, and data transformation.

The fundamental technique is the concept of service interface contracts, which play a critical role in allowing communication among the various components. The services are heterogeneous in both implementation language and operating environment, distributed geographically, and, possibly most important, commonly distributed in ownership. A main attraction of SOA is that it enables collaborations that cross company and organizational boundaries.

These characteristics of SOA aggravate the issue of program comprehension, which has been at the heart of the software maintenance problem for all previous generations of software. It is very important for a maintainer to understand software at a deep level before making changes; modifications made based on imperfect understanding are highly likely to fail, possibly with disastrous consequences. Such deep knowledge is, in practice, best obtained from skilled software engineers who have participated in the system's creation. Software maintenance typically becomes difficult and expensive precisely at the time when such individuals disperse, and take their expertise with them.

Most SOA composite applications are still young, so the challenges of SOA maintenance are still emerging. SOA implementations vary greatly so it is not clear that a typical SOA maintenance environment may look like. However, we hypothesize that SOA maintainers may face a situation similar to that is shown in **Figure 1**.

As illustrated in **Figure 1,** the maintainer works for one particular organization and typically has access to the services operated by that organization, which may be a mixture of components large and small, legacy and new, and implemented in different languages and styles. Further, these services interoperate with components owned and controlled by other organizations, and here the maintainer's information may be more fragmentary. The maintainer is unlikely to have, or desire, access to source code. Documentation may be partial and not always up-to-date. The one guaranteed-correct item will likely be the service interface description. If, as is often recommended, web services standards are being followed, then this description will take the form of a Web Services Description Language (WSDL) service interface and an Extensible Markup Language (XML) Schema Definition (XSD) for service datatypes. These documents must be current for the components to interoperate correctly at runtime.

In addition to source code, WSDLs and XSDs, there may be a plethora of other documents that shed light on how SOA composite applications actually work. These may include configuration files for the ESB and application servers in which the code is deployed, Business Process Execution Language (BPEL) orchestration instructions that tie component services together, project files describing the build process, and unstructured documentation in various formats; all of which may help provide insight into system operation.

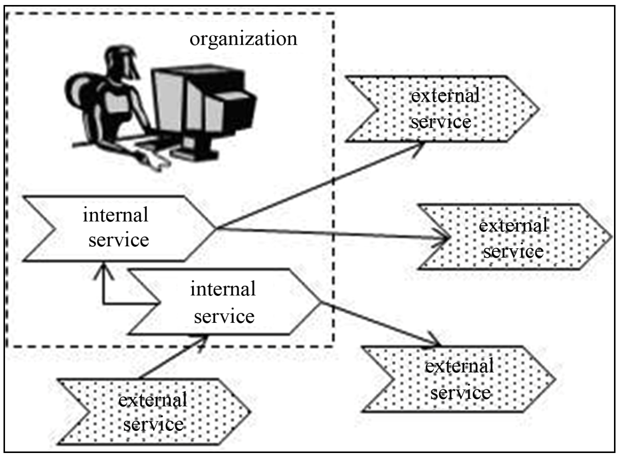


Figure 1. Typical SOA maintenance environment for a composite application.

In this environment, how will a SOA maintainer go about performing enhancements, adaptations or bug fixes? We hypothesize that the maintainer will follow advice that is as old as the software engineering profession: start by gathering all available information that is relevant to the problem [3]. That gathering process must include searches in the collection of documents described in the previous paragraph.

Our research group has been studying how intelligent search techniques could help ease the challenges of SOA maintenance. We have constructed a SOA search tool called SOAMiner—not as a finished product for immediate use—but rather as a research tool to explore the information needs of those maintaining SOA systems [4]. The research reported here builds upon and expands a two-part case study reported in Reichherzer [5] and White [6], which focused on observing software engineers as they use the search tool to help identify *what SOA maintainers will want to know*. This paper reports on additional analysis of the case study results and the development of methods and tools to enhance program comprehension for SOA systems.

From previous research results and related literature, it has become clear that naive search can be helpful, but for the combination of structured and unstructured SOA documents, it would be much more useful to combine simple text-based search with additional visualizations and abstractions of the information in the documents. More advanced search and visualization techniques will, for example, simplify the task of locating domain concepts within the system and perform impact analysis for proposed code enhancements.

In the remainder of this paper, we present our most recent research on enhancing program comprehension for SOA systems. Section 2 summarizes related research in the area of comprehension and maintenance of SOA systems. We then describe challenges in SOA comprehendsion and how intelligent search techniques and our search tool, SOA Miner, can contribute to improved understanding of those systems. Section 4 describes a case study that explores search within the context of maintenance for two different SOA applications, followed by the results, which highlight the need for specific abstractions that distill pertinent from irrelevant information. Section 5 presents additional analysis of the case study results as well as several visualization methods to enhance comprehension. Novel research reported in Sections 5 and 6 focuses on enhancing program comprehension for SOA systems through the development of various types of visualizations of the identified abstractions, including trees and Entity Relationship visualizations and a visualization tool, SOA Intel. The paper ends with a discussion of conclusions and plans for future work.

2. Related Work

A review of recent related work identifies the aspects of SOA that may make comprehension and maintenance even more difficult than it was with earlier systems [7-10]. Some of the main factors discussed are:

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

Researchers have also considered the organizational structures needed to support SOA maintenance. Kajko-Mattson, Lewis, and Smith [11] discuss how organizations adding SOA applications to their maintenance workload influence traditional IT roles. Several new roles emerge—which need to be filled—and new problems arise such as the prioritization of changes requested by different partners.

Recently, Papazoglou, Andrikopoulos, and Benbernou [12] described the problems of managing versions as a SOA system evolves. 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 should be constructed to identify the differences between an *as-is* currently deployed system and a *to-be* system with desired changes implemented. This suggestion highlights the importance of program comprehension since the creation of such a gap-analysis model would require a deep understanding of the services making up the *as-is* SOA system.

Our literature review identified only one paper that reports on experiences with program comprehension related to a specific interoperable system. Gold and Bennett [13] describe a research prototype web services system for integrating health care data from multiple providers. The project used the UK's National Health Service as an example. Their work suggests that there are many consequences which result from widespread distributed ownership of services. Maintainers will generally have to rely on WSDLs as descriptions of external services; however, WSDLs have many limitations with respect to comprehension since they are designed primarily to allow runtime calling of services. The maintainer will not be able to *drill down* into the code to get richer information since the code for an external service will most likely not be available to the maintainer. Gold and Bennett [13] argue further that since there are many service owners, the maintainer of a SOA composite application may need to deal with frequent unanticipated changes in external services with which he is interoperating. The maintainer will need to re-read updated WSDL interfaces and analyze the changes to identify necessary modifications to his own code. Time may be very short when there is a need to comprehend a modified service and make adjustments.

Little research has been conducted specifically on maintenance tools for Service Oriented Architecture systems. However, some researchers have proposed the use of dynamic analysis to aid in SOA understanding. 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. The Web Services Navigator tool, described by a group at IBM, collects trace data from a SOA application and provides five different views of the executing system [14]. Coffey, White, Wilde and Simmons [15] use dynamic analysis to address the problem of locating the message interchanges associated with a particular user feature in a SOA system. A rather different form of dynamic analysis involves hypothesizing an interface contract for an unfamiliar service and sending it a series of messages to confirm the correctness of the hypothesis [16]. Analysis of execution data can be a powerful way to gain understanding of software. However, dynamic analysis is not always practical because of the difficulty of collecting the necessary data when dealing with large systems running across many nodes.

3. SOA Search Support

3.1. SOA Artifacts

An alternative approach to analyzing systems dynamically is to analyze the artifacts that build them. In the case of SOA composite applications, SOA artifacts include a variety of different files that provide insight into the architecture of a SOA system and the messages exchanged between its services. The artifacts are automatically produced by SOA development and deployment tools and formatted according to standardized SOA Web languages, such as WSDLs, Simple Object Access Protocol (SOAP), XSD, and more. Since machines produce them from annotations in source code and user-specified deployment and integration information, the artifacts tend to be diverse and complex sometimes involving hundreds of lines of XML encoded descriptions that makes them difficult to inspect (the PAVER™ system described below has 10 WSDL files and 63 XSD files with a total of 49,000 lines and 14,000 lines of XML code respectively). Moreover, the SOA Web languages provide flexibility in the encodings of service and data descriptions, resulting in syntactic differences in SOA artifacts that further complicate their analysis. Yet, information that the artifacts contain is important to understand how a SOA composite application works. For example, the implementation of a service is described by a WSDL interface description, a type of contract that allows services to be loosely coupled. Among other things, the description includes operations of services, messages that may be exchanged between services along with data type information. Such information is valuable to software maintainers that will ultimately need to understand how services communicate and what data they exchange to make any changes to them consistent with the long-term maintenance goals of SOA composite applications.

3.2. Challenges in SOA Comprehension

Maintaining SOA composite applications requires a deeper understanding of the applications' services, their operations, and exchanged messages. To illustrate the maintenance challenges that engineers face, consider a few excerpts from the Web Auto Parts example discussed in more detail later. The artifacts include a BPEL file (Order Processing.bpel) that orchestrates multiple services to search inventories in the store for auto parts, computes taxes and shipping costs, among others, a WSDL file (Inventory Repository Artifacts.wsdl) that fronts the company's inventory database and a data type definition file (Inventory Query.xsd) that describes message data to and from the inventory repository. A maintainer that needs to know the data being passed when inventory is checked needs to step through a number of artifact files to answer this question, searching the files for text labels that match the names of services, operations, and data involved in the inventory check. A strategy to find the hidden information within the SOA artifacts may involve the following sequence of steps:

1) Examine the BPEL file to search for a partner service that involves inventory checks in the processing of an order.

2) Identify the partner link type associated with the partner service.

3) Look for the WSDL service interface file that implements the partner link type. This specific information is not available in the BPEL code. It must be discovered manually by matching tag names across the different WSDL files.

4) Find the operations associated with the service in the service interface file.

5) Identify the operation involved in the inventory check using the name of the operation and keywords for filtering.

6) Search for the correct input message specified by the operation. The input message contains the data type for the data passed to the service.

7) Find the data type within the same WSDL file or imported XSD files.

8) If the data type is a complex type, identify the parts of the type. This may involve additional searches for more data types.

In conclusion, multiple searches are needed through various SOA artifacts to answer a maintenance question. In the specific case of Web Auto Parts, service names, operations, and messages include keywords in their labels that made it possible to identify the correct data type. The discovery of such information requires an in-depth understanding of the languages encoding the SOA artifacts as well as their relationships.

3.3. SOA Miner: A Search Tool

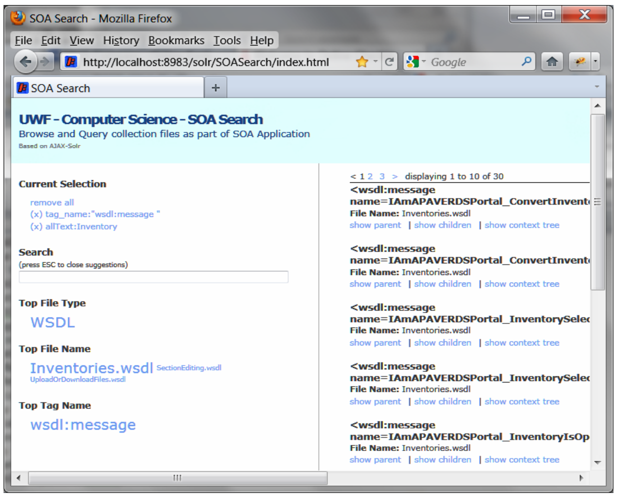
Scenarios such as the one described above lead us to believe that future SOA maintainers will need the ability to search quickly across multiple document types and to integrate multiple kinds of information as they build their conceptual picture of a composite SOA application. However, there is not yet any published experience about what actual types of searches SOA maintainers will want to perform.

To explore this question, we have developed a prototype SOA search tool called SOA Miner, as a research prototype to explore information needs. SOA Miner currently focuses on searches of documents having XML structure since WSDLs, XSDs, BPELs and many ESB configuration files take this form. SOA Miner is built on the Apache Solr search tool [17], with an interface derived from AJAX-Solr [18]. To use SOA Miner, the software engineer indexes the collection of files that have an XML structure and then queries the collection 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 are displayed on the right.

4. Two-Part Case Study of SOA Search

4.1. General Structure

We performed a two-part case study of SOA systems to


SOA Search - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:8983/solr/SOASearch/index.html
Google
SOA Search

UWF - Computer Science - SOA Search
Browse and Query collection files as part of SOA Application
Based on AJAX-Solr

Current Selection
remove all
(x) tag_name:"wsdl:message "
(x) allText:Inventory

Search
(press ESC to close suggestions)

Top File Type
WSDL

Top File Name
Inventories.wsdl SectionEditing.wsdl UploadOrDownloadFiles.wsdl

Top Tag Name
wsdl:message

< 1 2 3 > displaying 1 to 10 of 30

<wsdl:message name=IAmAPAVERDSPortal_ConvertInvent
File Name: Inventories.wsdl
show parent | show children | show context tree

<wsdl:message name=IAmAPAVERDSPortal_ConvertInvent
File Name: Inventories.wsdl
show parent | show children | show context tree

<wsdl:message name=IAmAPAVERDSPortal_InventorySelec
File Name: Inventories.wsdl
show parent | show children | show context tree

<wsdl:message name=IAmAPAVERDSPortal_InventorySelec
File Name: Inventories.wsdl
show parent | show children | show context tree

<wsdl:message name=IAmAPAVERDSPortal_InventoryIsOp
File Name: Inventories.wsdl
show parent | show children | show context tree

Figure 2. SOA Miner screenshot.

gain insight into what SOA maintainers would want to know as they try to understand a SOA composite application and perform a maintenance task. The objectives and methods of the two parts were the same, but since SOA systems are very diverse, we found two different composite applications representing different implementtation approaches to services-based computing. The case study thus involves two different "units of analysis" within a single context as described by Runeson and Host [19].

The main research question for the case study was: What abstractions would SOA maintainers find useful to help them understand a composite application?

The method for each part of the study followed the same steps. We started from a non-trivial services-based system and indexed artifacts from it such as WSDLs and XSDs. We also developed a list of questions to answer about the specific SOA application. The questions were loosely based on the search questions that Sim, Clarke, and Holt [20] found that software engineers asked when maintaining traditional software systems. The questions included both concept location queries that look for domain concepts (e.g., "What services/operations/messages deal with parts inventory?") and impact analysis queries (e.g., "If I change this data type, what code is affected?").

Case study participants were first provided some training in using the tool. Then they used it individually to respond to the questions while "thinking out loud" to verbalize their thought processes. An observer noted the searches they made and the areas where they seemed to encounter difficulties. Finally, for each part of the study, there was a group debrief session with all participants to discuss the results and contribute suggestions for improved search and visualization support.

4.2. The Web Auto Parts System

The first unit of analysis involved a hypothetical online automobile parts dealer called Web Auto Parts [5]. Web Auto Parts models an Internet start-up company that is using SOA for rapid development. Its software uses BPEL for orchestration of commercially available external services from well-known vendors. As shown in **Figure 3**, the Order Processing workflow for Web Auto Parts has two stubbed in-house BPEL services (Order Processing and Inventory Repository) and four comercially available external services:

- Amazon Web Services-*Simple DB* (data base) and *Simple Queue Service* (message queuing).
- StrikeIron.com-*Tax Data Basic* (sales tax rates).
- Ecocoma-*USPS* (shipping costs).

Web Auto Parts is, of course, much smaller than most real SOA applications as shown in **Table 1**. However, it is useful for a case study since it consists of syntactically correct BPEL code and contains XSD and WSDL documents typical of current industrial practice.

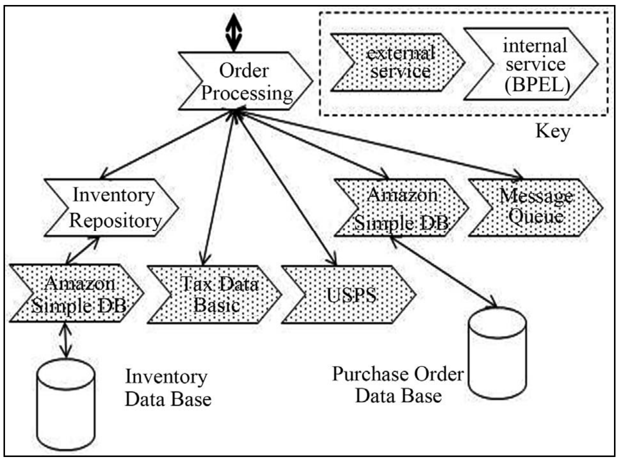


Figure 3. Web Auto Parts-services in the order processing workflow.

Table 1. SOA artifacts of web Auto Parts.

Filename	Lines of Code
wsdl.AmazonSimpleDB	611
xm.ldeploy	51
xsd.InventoryQuery	28
wsdl.InventoryRepositoryArtifacts	69
bpel.OrderProcessing	118
bpelex.OrderProcessing	55
xsd.PurchaseOrder	36
dlws.QueueService	1043
wsd.5lTaxDataBasic	436
wsdl.usps	197
Total	2644

Study participants were three computer science faculty members who were familiar with the basic concepts of SOA. However, only one of them had any SOA programming experience or experience with WSDLs, XSDs and BPEL. They were given only very general information about Web Auto Parts and about the Order Processing workflow, typical of what a maintainer might have in dealing with an unfamiliar application.

4.3. The PAVER™ System

The second unit of analysis involved a full-scale industrial SOA application called PAVER™ [6]. It included a graduate student and professional software engineers experienced in services computing. PAVER™ is a pavement management system for condition-based maintenance management of airport pavements and roadways.

The version used in this study had 10 services, which totaled approximately 400,000 lines of Visual Basic code. PAVER™ can be used by pavement engineers in several ways: 1) to develop an inventory of the pavements to be managed; 2) to collect distress data about inventory items using an international standard for such field observations; 3) to model pavement condition over time, and 4) to plan budgets for repair work in order to make the best use of limited resources.

The PAVER™ system is not a typical SOA application because it uses SOA principles to manage the separation of what appears to the user as a unified system into a family of components that are mostly independent yet closely cooperate. The system uses Microsoft's Windows Communication Foundation (WCF) to implement a family of cooperating services that span the major modules within PAVER™. The WSDL and XSD files are generated automatically from interface definitions using specialized WCF tags, and can be used to describe the services in a standard way.

The case study followed the same structure described in Section 4.1, with pre-indexing of the WSDLs and XSDs, a training exercise for the participants, a main study using a questionnaire, and a debrief session. However, the main study was structured somewhat differently since the study dealt with a real application and some participants were professional software engineers having experience with that application. The main study took place in three phases as follows:

- First, a graduate student with some SOA background but no PAVER™ experience went through the questions in a university setting.
- Second, a software engineer at the company that develops PAVER™ worked on the questions and related SOA Miner's responses to his insight from developing the PAVER™ code.
- Third, two software engineers from the PAVER™ development company used SOA Miner searches and compared the results with information obtainable from Microsoft's Visual Studio™ programming environment used in their daily work. They also experimented with allowing Visual Studio to generate a stub client for one of the services to see what information was recoverable in that way.

4.4. Case Study Results: Identified Abstractions

Almost all search tools present a low-level *worm's eye* view of the subject matter. Just as a source code search tool may identify many isolated lines that match search criteria, SOAMiner finds many isolated XML tags. To make changes successfully, a maintainer needs to create for himself a higher-level conceptual view of the software, and this may require many searches. We would like to supplement search results with higher-level *abstractions* to speed up the maintainers work. For example, a click on a search result could provide a pop-up showing how that isolated result fits into a larger picture as shown in **Figure 4**.

The case study identified a variety of such abstractions. They could be classified roughly as *service linkage abstractions* that pull together distant information across service boundaries, and *summarizing abstractions* that eliminate verbose syntax that may hinder understanding. A third category, *datatype abstractions*, has characteristics of both categories and, because of its importance, is discussed separately in Section 5.

The service linkage abstractions summarize logical chains in XML, which are found by tracing names from one XML element to another. Case study participants identified several relationships between service definitions and service invocations that they felt would help answer questions such as: *How do services call each other*?

High level-service invokes service. This abstraction would summarize chains similar to:

<bpel: process> (BPEL file)
<bpel: partnerLink> (BPEL file)
<plnk: partner Link Type> (WSDL file)
<service> (WSDL file)

This abstraction essentially ties a BPEL process to the services that it may use since a partner link of the correct type has been declared. It does not establish if or where such usage actually takes place. However the abstraction can provide a high-level view of the potential relationships in a composite application and supports further exploration of input and output data.

Low level-process invokes operation. Summarizes chains similar to:

<bpel: invoke> (BPEL file)
<bpel: partner Link> (BPEL file)
<plnk: partner Link Type> (WSDL file)
<port Type> (WSDL file)

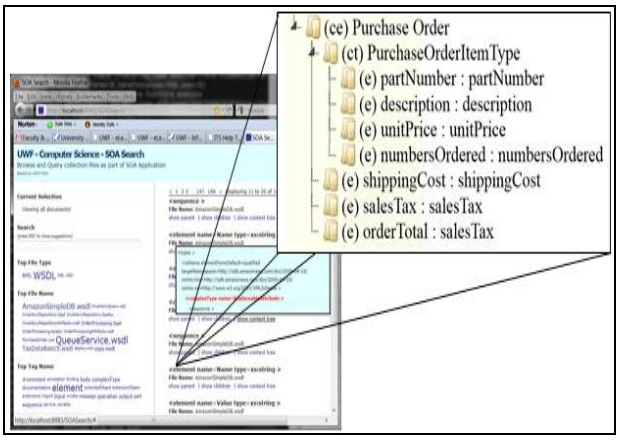

(ce) Purchase Order
(ct) PurchaseOrderItemType
(e) partNumber : partNumber
(e) description : description
(e) unitPrice : unitPrice
(e) numbersOrdered : numbersOrdered
(e) shippingCost : shippingCost
(e) salesTax : salesTax
(e) orderTotal : salesTax
UWF • Computer Science • SOA Search

Figure 4. Displaying an abstraction in a search result.

<operation> (WSDL file)
<bpel: receive> (BPEL file of invoked service)

This abstraction ties a BPEL invocation of an operation on a service to the high-level WSDL description and through that to a BPEL implementation of the called operation. This particular chain enables the maintainer to examine the actual code that may be implementing a called service.

Low level—"*called by*". Summarizes chains similar to:
<bpel: receive> (BPEL file of invoked service)
<bpel: partner Link> (BPEL file of invoked service)
<bpel: partner Link> (BPEL file of calling service)
<bpel: invoke> (BPEL file of calling service)

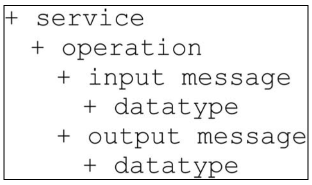
This abstraction allows a maintainer to understand how other BPEL processes may invoke a particular operation implemented in BPEL.

Summarizing abstractions are needed because XML is so verbose and because WSDL and XSD files are often machine generated, making them still more verbose. All the study participants noted that the WSDL/XSD description of a service was too dispersed and hard to navigate. In debrief sessions they all agreed that one solution would be a simple expandable tree abstraction of a service as shown in **Figure 5**. This eliminates many XML tags and provides a more compact description that should be adequate for many purposes.

The best example of machine generated verbosity identified in these studies comes from WCF generated XSD files. **Figure 6** shows how WCF has generated five tags encapsulating an empty <sequence>, simply to convey the information that the response to a Report Delete operation is empty. The single word "void" could replace the whole structure.

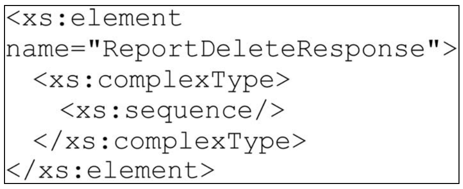
5. Datatype Abstraction and Visualization

This research analyzes and expands upon the results of the case study described above to facilitate enhanced program comprehension for SOA systems. The case study results showed that naive search can be helpful, but for the combination of structured and unstructured SOA documents, it would be much more useful to combine simple text-based search with additional visualizations and abstractions of the information in the documents. As a result, more advanced search and visualization techniques were developed to simplify the task of locating domain concepts within the system and of performing impact analysis for proposed code enhancements. More specifically, the case study results demonstrated the need for specific *abstractions* that distill pertinent from irrelevant information and for *visualization*s to display the pertinent information to software maintainers. This section presents methods that we have developed to enhance SOA program comprehension, including an enhanced search and visualization tool, and tree and Entity Relationship (ER) visualizations of the identified abstractions.



```
+ service
  + operation
    + input message
      + datatype
    + output message
      + datatype
```

Figure 5. Expandable tree abstraction of a service.



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

Figure 6. Datatype description of a void return type.

5.1. Analysis of the Case Study Results

The results of the case study highlighted the challenge in understanding SOA datatypes and their significance in SOA program comprehension. Experienced study participants emphasized the importance of understanding the data model underlying a SOA application before making any changes to it. The results indicated that while searches for services and for operations went fairly smoothly, searches related to datatypes were substantially more difficult. Data definition information is spread across multiple tags, such as <message> and <part> tags in WSDLs, and <element>, <complexType> and <sequence> tags in XSDs. Since a search result shows individual matching tags, the user often has to make multiple searches to identify the entire structure. It is not even easy for SOA maintainers to know in which files to search for datatype information since service authors may choose to reference separate XSD files or to incorporate such information directly into the <types> section of the WSDL itself.

Additionally, the results suggested that it was easy to make time-consuming mistakes in trying to locate datatype information. For example in trying to find the data passed in an inventory query, participants searched for a <complexType> tag matching the string "inventtoryQuery". However, in an XSD, the datatype can be named in either an <element> tag as depicted in **Figure 7**, or in a <complexType> tag. Much time may be lost if the maintainer searches on the wrong alternative.

Due to the variety of difficulties identified with datatypes, a high priority for enhanced SOA program comprehension is to provide a compact way of abstracting and visualizing them. An abstraction would be constructed by walking the XML structure and collecting references to each datatype and the elements and types it contains. For visualizing the resulting abstraction, we

```
<element name="inventoryQuery">
  <complexType>
    <sequence>
...
```

Figure 7. Datatype named in the <Element> tag.

employ two strategies, a tree view and an ER view, each with its advantages and disadvantages.

5.2. Tree Visualization of Datatypes

A tree visualization would use an expandable tree view of datatypes. For example, consider a simple XSD element from the <types> section of the Amazon Simple DB WSDL file. Select Response is the payload of the Select Response Msg, which is the result of the Select operation in Amazon Simple DB. The XSD definition of Select Response is:

<xs:element name = "Select Response" >
<xs:complexType >
<xs:sequence>
<xs:element ref = "tns: Select Result"/>
<xs:element ref = "tns: Response Metadata"/>

Here we see that the element Select Response has two child elements that reference Select Result and ResponseMetadata so we would next have to locate these elements. If we continue this location process in total we would need to identify five separate elements scattered throughout Amazon Simple DB WSDL.

As a first visualization alternative, we could display a compact expandable tree view for Select Response:

CES-Select Response
+E-element-Select Result
+E-element-Response Metadata

This view uses abbreviations to summarize commonly occurring structures. "CES" denotes a "complex element sequence" and "E" an "element". The view has "+" symbols next to the child elements indicating that they can be expanded. In a user interface, a click on the "+" would expand the tree to show additional information. This allows dependencies for an element to be aggregated and displayed without requiring additional queries by the user. The fully expanded tree view for Select Response is as follows:

CES-Select Response
-E-element-Select Result
CES-Select Result
-E-element-Item
CTS-Item
E-element-Name
-E-element-Attribute
CTS-Attribute
E-element-Name
E-element-Value
E-element-Next Token
-E-element-Response Metadata
CES-Response Metadata
E-element-RequestId
E-element-Box Usage

However, for a complete visualization of a datatype it would be useful to show how it is used; a right-click or other gesture could show a list of the elements that depend upon it, as shown below. For example, ResponseMetadata is utilized by additional XSD elements besides Select Response. In fact, in this particular WSDL, Response Metadata is a common element of all "response" elements. The enumeration of these dependencies, listed below, provides additional invaluable information about the layout of SOA artifacts.

Response Metadata References:
-Select Result
-Batch Delete Attributes Response
-Delete Attributes Response
-Get Attributes Response
-Batch Put Attributes Response
-Put Attributes Response
-Delete Domain Response
-Domain Metadata Response
-List Domains Response
-Create Domain Response

5.3. Entity Relationship Visualization of Datatypes

An alternative to the tree view involves generating an Entity Relationship (ER) view of SOA datatypes. ER diagrams present an abstraction of a data model composed of entities, relationships and attributes. ER diagrams show how entities, which contain attributes, are related, but are not limited to a hierarchical structure. Since not all relationships between data elements are hierarchical in nature or necessarily have a tree structure, the ER model is an effective way of showing all relationships between entities, and has become an established method of presenting relationships between data elements or entities [21].

Entities or complex types (in XML), are depicted by rectangles in ER visualization and are units used to hold data or concepts of data. Characteristics of complex types are referred to as attributes in ER visualization and are depicted in ER diagrams as ovals attached to the entities. Relationships between the entities, denoted by element refs (in XML), are depicted by diamonds in ER diagrams. ER diagrams also show how many of one element is related to another element. For example, as shown in **Figure 8**, we can see that one Select Result element can be related to many Items, or one Item can be related to many Attributes. And we can also see immediately that Response Meta Data is related to several other entities, not shown in the figure for the sake of brevity.

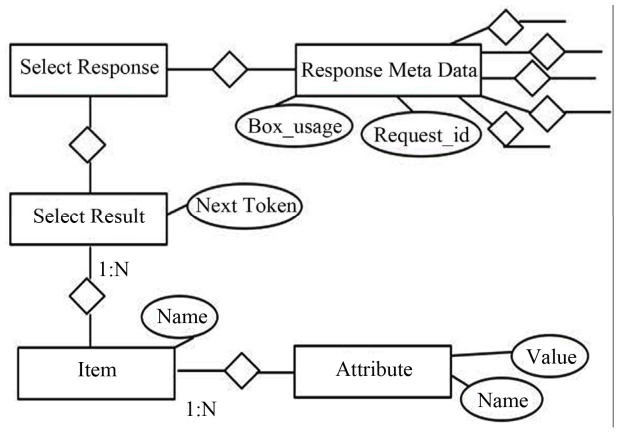


Figure 8. ER visualization of the select response element.

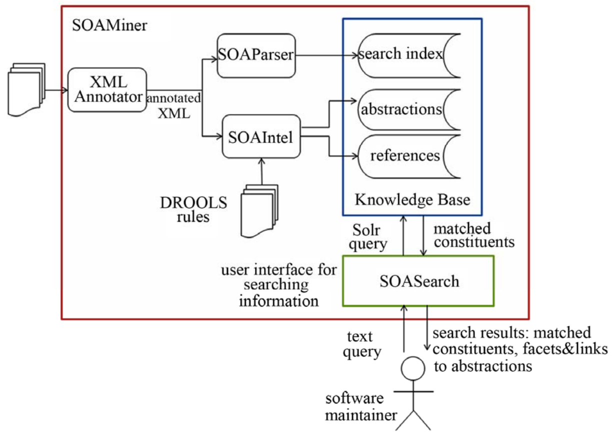


Figure 9. SOA Miner and SOA Intel architecture.

The expandable tree visualization has the advantage of being more compact than an ER diagram and it has a textual representation that could be easier to display in a web browser or similar interactive interface. On the other hand the ER visualization makes use of a well-known modeling method that will be familiar to many software engineers and perhaps conveys more information without the need for mouse clicks or other navigational gestures.

6. Enhancing SOA Search with SOA Intel

Modern search technology can help users quickly identify pieces of information whether on the Web or in documents including SOA artifacts. Search can support program comprehension of complex, heterogeneous SOA systems by discovering patterns and links across the various types of documents that compromise SOA composite applications. Research involving SOA Miner has demonstrated the usefulness of exposing information in SOA artifacts to software engineers that can help with maintenance activities. However, search tools can only discover and present patterns as they exist in the documents themselves encoded in the language of the indexed documents. As the case study results have shown, what maintainers need to know most is conceptual information about the composite application, not syntactic information included in the SOA artifacts. They need to know what services are involved in a composite application and what data is being exchanged among services. Thus, while search is useful, we need to be able to transform the search results into meaningful entities. The identified abstractions discussed earlier represent those meaningful units.

Compiling abstractions from the diverse artifacts of SOA composite applications requires interpretation and knowledge of the language that encodes the artifacts. A difficulty is that in the SOA open environment, the relevant abstractions will vary from system to system and over time as standards, practices and tools change.

Thus, we need a flexible approach to complement SOA search and automate the discovery of abstractions within the SOA artifacts. An ideal tool would index the collection of artifacts from a composite application and provide maintainers with two types of information:

1) Abstractions where it can discover them, or

2) Text snippets taken from the artifacts where it cannot.

The tool would also allow for the definition of additional abstractions so that more and more search results can be of the first category. Such a tool would have to be flexible to adapt to a wide range of SOA artifacts from different environments and allow for the inclusion of new abstractions as they are identified by software engineers.

In an effort to extend SOA Miner with the kinds of search capabilities identified from an analysis of the case study results outlined above, we developed SOA Intel, a tool to discover abstractions within SOA artifacts. **Figure 9** shows the relationship between SOA Miner, SOA Intel, the SOA artifacts and software maintainers. As shown in the figure, a software maintainer submits a text query using the SOA Search interface. SOA Intel then uses knowledge-based systems methods and applies a rule-based approach to build abstractions from the artifacts. Next, SOA Miner indexes these abstractions to provide them to users upon request. A reasoning engine automates the process of the expert's analysis of SOA artifacts by executing chains of rules on the artifacts once they are committed to the engine's working memory. Matching rules build and store the abstractions in working memory, which can then lead to the discovery of additional abstractions and relationships between them. Finally, after all rules have fired, working memory may be queried to collect the abstractions and store them for future retrieval by SOA Miner in response to user-entered search queries. Each abstraction is formatted as an XML snippet that includes constituents and relations from the SOA artifacts to model the abstraction. More details on the rules can be found in [22].

Rule engines have traditionally been used to capture expert problem solving as a set of rules and apply them to reason about a solution for a new problem. Through experiments and case studies involving domain experts, we can create a set of rules that identify abstractions within the SOA artifacts, and extract and transform them into human-readable representations. In essence, the rules are designed to capture an expert's experience with identifying useful excerpts of information relevant to maintenance problems.

7. Discussion and Conclusions

SOA offers the prospect for greater interoperability among software systems by enabling developers to configure software as services and to interconnect services with other services independent of differences in operating platform and programming and communicating languages. Resulting systems can bridge heterogeneous environments of platforms and languages across institutional boundaries. However, the new languages, infrastructures and flexibility that accompany SOA also introduce new complexities.

From the perspective of software maintenance, SOA systems require:

1) An in-depth understanding of the mechanics involved in the deployment of composite applications and their interaction;
2) Expertise in many different languages and environments that make up a SOA composite application, and
3) Insight into the distribution and ownership of services.

These new challenges make it difficult for maintainers to determine how an existing SOA system actually works. We believe that searching techniques can help maintainers locate needed information in the large body of artifacts that describe a SOA application. However, search will need to be accompanied by a process of abstraction and visualization to enhance the comprehension of search results. Additional analysis of the case study results identified some of the abstractions and visualizations needed to enhance SOA program comprehension. These were classified as service linking abstractions to identify relationships between services, summarizing abstractions to remove unneeded verbose syntax, and data type visualizations to help understand the data that services share. These abstractions and visualizations distill useful information by compiling dependencies among services and references of messages to data types into single, coherent units free from syntactic overhead of XML encoded machine descriptions. Several visualizations were developed, including trees, Entity Relationships and the framework for an intelligent search-based visualization tool.

Several directions for future work can be explored to provide further support for program comprehension of SOA systems. We are exploring methods for semantic analysis of SOA composite applications, including concept maps [23] and ontologies. Future tools to support SOA maintainers could build on ontologies describing the web services standards, the most common extensions to these standards, and domain concepts from different problem domains such as health care, travel, etc. Information from the ontologies could be used in several ways. It could provide searchable help that a SOA maintainer could use in interpreting the WSDL, XSD, and other documents encountered in his work. It could also provide input for search tools such as SOA Miner for synonym analysis of queries and help prioritize query results.

Research into program comprehension for SOA systems is still at a very early stage and doubtless, many novel tools and techniques remain to be developed. However, as an initial step, we think that search enhanced by abstraction and visualization can make an important contribution in addressing the emerging challenges of SOA maintenance.

8. 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 SOA Miner, and Ben Hartmann and Mircea Manea, Software Engineers at Intelligent Information Technologies, for their participation in the PAVER™ case study.

REFERENCES

[1] N. Josuttis, "SOA in Practice: The Art of Distributed Software Design," O'Reilly Media, Sebastopol, 2007.

[2] G. Lewis, E. Morris, S. Simanta and D. Smith, "Service Orientation and Systems of Systems," *IEEE Software*, Vol. 28, No. 1, 2011, pp. 58-63. doi:10.1109/MS.2011.15

[3] S. D. Fay and D. G. Holmes, "Help! I Have to Update an Undocumented Program," *Proceedings of the IEEE Conference on Software Maintenance*-1985, Washington DC, 11-13 November 1985, pp. 192-204.

[4] L. White, T. Reichherzer, J. Coffey, N. Wilde and S. Simmons, "Maintenance of Service Oriented Architecture Composite Applications: Static and Dynamic Support," *Journal of Software Maintenance and Evolution*: *Research and Practice*, Vol. 25, No. 1, 2011, pp. 97-109. doi:10.1002/smr.568

[5] T. Reichherzer, E. El-Sheikh, N. Wilde, L. White, J. Coffey and S. Simmons, "Towards Intelligent Search Support

for Web Services Evolution: Identifying the Right Abstractions," *Proceedings of* 2011 13*th IEEE International Symposium on Web Systems Evolution* (*WSE*), Williamsburg, 30 September 2011, pp. 53-58. doi:10.1109/WSE.2011.6081819

[6] L. White, N. Wilde, T. Reichherzer, E. El-Sheikh, G. Goehring, A. Baskin, B. Hartmann and M. Manea, "Understanding Interoperable Systems: Challenges for the Maintenance of SOA Applications," *Proceedings of the* 45*th Hawaii International Conference on System Sciences* (*HICSS*), Maui, 4-7 January 2012, pp. 2199-2206.

[7] G. Canfora and M. Di Penta, "New Frontiers of Reverse Engineering," *Proceedings of the* 29*th International Conference on Software Engineering*, Minneapolis, 20-26 May 2007, pp. 326-341. doi:10.1109/FOSE.2007.15

[8] N. Gold, C. Knight, A. Mohan and M. Munro, "Understanding Service-Oriented Software," *IEEE Software*, Vol. 21, No. 2, 2004, pp. 71-77. doi:10.1109/MS.2004.1270766

[9] K. Kontogiannis, "Challenges and Opportunities Related to the Design, Deployment and Operation of Web Services," *Proceedings of the* 24*th Conference on Software Maintenance*, Beijing, 28 September-4 October 2008, pp. 11-20. doi:10.1109/FOSM.2008.4659244

[10] G. A. Lewis and D. B. Smith, "Service-Oriented Architecture and Its Implications for Software Maintenance and Evolution," *Proceedings of the* 24*th Conference on Software Maintenance*, Bejing, 28 September-4 October 2008, pp. 1-10. doi:10.1109/FOSM.2008.4659243

[11] M. Kajko-Mattsson, G. A. Lewis and D. B. Smith, "Evolution and Maintenance of SOA-Based Systems at SAS," *Proceedings of the* 41*st Annual Hawaii International Conference on System Sciences* (*HICSS*), Waikoloa, 7-10 January 2008, p. 119. doi:10.1109/HICSS.2008.154

[12] M. P. Papazoglou, V. Andrikopoulos and S. Benbernou, "Managing Evolving Services," *IEEE Software*, Vol. 28, No. 3, 2011, pp. 49-55. doi:10.1109/MS.2011.26

[13] N. Gold and K. Bennett, "Program Comprehension for Web Services," *Proceedings of the* 12*th IEEE International Workshop on Program Comprehension*, Bari, 24-26 June 2004, p. 151.

[14] W. De Pauw, M. Lei, E. Pring, L. Villard, M. Arnold and J. F. Morar, "Web Services Navigator: Visualizing the Execution of Web Services," *IBM Systems Journal*, Vol. 44, No. 4, 2005, pp. 821-845. doi:10.1147/sj.444.0821

[15] J. Coffey, L. White, N. Wilde and S. Simmons, "Locating Software Features in a SOA Composite Application," *Proceedings of the* 8*th IEEE European Conference on Web Services* (*ECOWS*'10), Ayia Napa, 1-3 December 2010, pp. 99-106. doi:10.1109/ECOWS.2010.28

[16] S. Halle, T. Bultan, G. Hughes, M. Alkhalaf and R. Villemaire, "Runtime Verification of Web Service Interface Contracts," *Computer*, Vol. 43, No. 3, 2010, pp. 59-66. doi:10.1109/MC.2010.76

[17] Apache Software Foundation, "Apache Solr," 2011. http://lucene.apache.org/solr/

[18] GitHub Inc., "Evolvingweb/AJAX-Solr," 2012. http://github.com/evolvingweb/ajax-solr

[19] P. Runeson and M. Host, "Guidelines for Conducting and Reporting Case Study Research in Software Engineering," *Empirical Software Engineering*, Vol. 14, No. 2, 2009, pp. 131-164. doi:10.1007/s10664-008-9102-8

[20] S. E. Sim, C. L. A. Clarke and R. C. Holt, "Archetypal Source Code Searches: A Survey of Software Developers and Maintainers," *Proceedings of the* 6*th International Workshop on Program Comprehension* (IWPC'98), Ischia, 26 June 1998, pp. 180-187. doi:10.1109/WPC.1998.693351

[21] S. Bagui and R. Earp, "Database Design Using Entity-Relationship Diagrams," 2nd Edition, Auerbach Publications, Boca Raton, 2012.

[22] G. Goehring, T. Reichherzer, E. El-Sheikh, D. Snider, N. Wilde, S. Bagui, J. Coffey and L. White, "A Knowledge-Based System Approach for Extracting Abstractions from Service Oriented Architecture Artifacts," *International Journal of Advanced Research in Artificial Intelligence*, Vol. 2, No. 3, 2013, pp. 44-52.

[23] J. Novak and D. Gowin, "Learning How to Learn," Cambridge University Press, New York, 1984. doi:10.1017/CBO9781139173469