# Interoperable Systems and Software Evolution: Issues and Approaches

**Norman Wilde, Sikha Bagui, John Coffey, Eman El-Sheikh, Thomas Reichherzer, Laura White, George Goehring, Chris Terry**

University of West Florida, Pensacola, Florida, U. S. A.

**Arthur Baskin**

Intelligent Information Technologies, Indianapolis, Indiana, U. S. A.

*Interoperability is essential for modern enterprise software; one of the most promising ways of providing interoperability is though Services Oriented Architectures (SOA) usually implemented using the Web Services (WS) standards. SOA/WS has the potential to be a transformational technology but there are a number of problems that may hinder its application. One of these is the classic slowness of software evolution. This paper discusses the issues of SOA evolution and describes ongoing research experimenting with the use of search technology to speed comprehension of SOA applications. Flexible but specialized search tools may be a good match for the "open world" of a SOA system which may encounter frequent novelties in programming languages and technology during its lifetime. We describe a basic search tool adapted to SOA/WS artifacts, a knowledge-based extension to it to improve software comprehension, and ongoing work to handle additional document types and to provide ontology-based support. Development of support tools for SOA evolution could be a fruitful topic for industry-university collaboration. Such tools would be an enabler for the interoperable information systems needed to do business in the modern world.*

## 1 Introduction

Two trends in business and government drive the growing need for interoperable information systems:

1. As companies form partnerships and governments strive to integrate the work of different departments (Janssen et al. 2011), business processes become ever more interconnected even across organizational boundaries.

2. Each step in such processes depends ever more heavily upon software support, usually involving pre-existing information systems developed using diverse standards and technologies.

It is clearly unrealistic to design a new technologically harmonized information system to meet each emerging need. So the only practical solution is to find ways to allow existing, technologically diverse systems to interoperate. Interoperation has been described as having two or more independent systems operate in a coordinated and meaningful fashion such that processes are effectively merged or information is effectively shared (Scholl and Klischewski 2007).

While there have been many attempts to achieve interoperability, the approach that seems to have the most followers at the moment is Services Oriented Architecture (SOA), usually implemented following the Web Services (WS) standards (Josuttis 2007 Chapter 16). SOA is not regarded as a specific architecture but rather as a general way of structuring software. Terminology varies but typically *composite applications* are constructed by *orchestrating* loosely coupled *services* running on different nodes and communicating via message passing. Ideally each service represents a discrete business function that can act as a reusable component across multiple applications. 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.

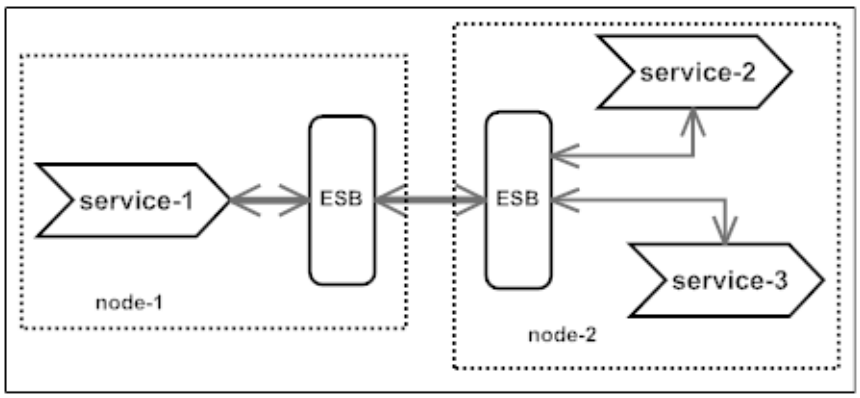


Fig. 1 - Structure of a SOA Composite Application

SOA/WS has the potential to be a transformational technology with widespread impact on the way humans live. Earlier transformational technologies, such as printed books or steam-powered transportation, broke down barriers that prevented connections; in the one case connections of ideas from human to human, in the second connections of goods between producers and consumers. As connections become feasible, interactions multiply and new human opportunities emerge.

In theory, SOA/WS may become transformational if it can enable connections between data and processing, both within and across organizational boundaries. However experience with SOA/WS has been somewhat mixed. Both organizational and technical obstacles have sometimes made it difficult to achieve the promised benefits of this approach (Luthria and Rabhi 2012). In this paper we will discuss one such obstacle that is one of the oldest problems in Software Engineering, that is, the cost and delays of software evolution.

A distinguishing characteristic of interoperable composite applications is that ownership and control over services is also often distributed. Whether the different services are contained within one large organization (such as the many agencies within a government) or whether some are completely outsourced (as with commercial cloud services), the IT manager must deal with components that are not under his control. Evolution may then be driven by external changes on an inflexible schedule.

> For example one of the authors of this paper is the developer of a composite application that gathers price data from several sources, including the Amazon Product SOAP API to prepare a pricing guide for a retail business. Recently Amazon has announced that, in a few months' time, this interface will be restricted to Amazon affiliates only. The developer must now evaluate where and how he is using that particular service and develop an alternative or work-around.

If IT managers are to commit to interoperable systems, they must first have confidence that they can respond to such external changes with agile software evolution. In this paper we will discuss some of the technical challenges in the evolution of interoperable applications and describe our ongoing research into support tools based on enhanced search technology for SOA/WS.

## 2 Software Evolution and Interoperability

The term *maintenance* has traditionally been applied to all changes that are performed upon a software system after its deployment. However since most changes are not bug fixes, some have proposed that we should instead call this process *software evolution* and this may indeed be a better term to describe a system's overall history of change. However "evolve" is a passive verb so it misses the fact that software change requires work by talented and costly professionals. Perhaps a compromise would be to say that the software *evolves* because of the work Software Engineers perform to *maintain* it and so we will use both terms in this paper.

The defining characteristic of software maintenance/evolution as opposed to development is that any proposed change needs to take into account a large base of existing software. The change is thus highly constrained; a Software Engineer cannot just look for the best way to implement a requirement but rather must look for the best way given the existing design and code base. Changes are costly and hard to plan because of uncertainties in the time required to understand the exist-

ing system, plan a change, identify impacts of the change and test that all impacts have been correctly handled. It is not uncommon for even experienced Software Engineers to grossly underestimate the ripple effects of a software change.

Several authors have pointed out that SOA composite applications may present a particular challenge for program comprehension and thus for software maintenance (Gold et al. 2004, Canfora and Di Penta 2007, Lewis and Smith 2008, Kontogiannis 2008). Most of the factors mentioned stem from or are exacerbated by the distributed ownership of the different services:

1. Services making up a composite application may use diverse technologies: in their operating system, in their programming languages, and in their messaging layer. This diversity complicates invocation across system boundaries. The Web Services standards attempt to mitigate these difficulties by prescribing the use of Web Services Description Language (WSDL) to specify service interfaces with XML Schema Descriptions (XSDs) used to describe data passed in messages. Given a WSDL and its XSDs, tools can construct proxies and stubs allowing code using one technology to invoke code using another. However the tools are not currently problem free (McGregor et al. 2012) and in any case the WSDL really only covers the mechanics for message exchange. It does not provide any information about how service invocations need to be sequenced, the state changes produced by a service invocation, and other semantic factors that may be essential for program comprehension.
2. The services may employ different semantics, especially subtly conflicting meanings of data items, which makes it problematic for one system to use data provided by another. For example, Gold and Bennett mention that the term "child" has numerous different meanings in different organizations in the United Kingdom health care domain (Gold and Bennett 2004).
3. Composite applications may face complex and changing security requirements: some operations may be restricted, data may need to be confidential and all actors in the system should not have the same access. This means that maintenance changes need to be looked at carefully not only from the point of view of their functionality, but also as to their application-wide effects on security.
4. For commercial reasons the owners of a service may not choose to make available complete documentation of design, defects encountered, change history, and so on. Software Engineers in the consumer organization may thus have greater difficulty understanding changes in service behavior.

## 3 Program Comprehension for SOA Evolution

Thus while interoperability is a business necessity, it brings with it increased complexity and some level of loss of control. The challenge for SOA evolution then is to perform needed changes quickly and correctly despite these new factors.

The key roadblock is the same as for evolution of traditional systems: a Software Engineer has to understand a program in order to change it effectively and safely.

An obvious goal then is to reduce the costs of SOA program comprehension, specifically by providing tools and techniques that can help Software Engineers navigate and understand the artifacts making up a composite application. This goes beyond studying code to analyze artifacts such as:

1. Descriptions of both external and internal services; WSDLs and XSDs together with documentation in any format provided by the owner.
2. Deployment configuration files such as the *web.config* of ASP.NET, the *web.xml* of J2EE and numerous others that determine how services are accessed, how the execution environment is configured, what security restrictions are set, etc.
3. Middleware configuration, such as configuration files for a particular ESB, application server-specific configuration such as *sun-web.xml* (for GlassFish) etc.

Clearly one challenge is the "open world" nature of SOA (Van den Heuvel et al. 2009). Many of the artifacts mentioned are specific to a particular technology or vendor. Yet technologies, vendors, and the specific set of partners in a composite are likely to change greatly over the application's life cycle. We cannot predict an exact static mix of technologies, programming languages, and documentation formats that will remain valid for the life of any particular composite application. Indeed, it is likely that maintenance of each mature SOA composite application will involve a somewhat different and changing combination of artifacts.

This means that support tools for SOA evolution must also be able to function in this open world. Somehow we need to create support tools that will have the flexibility to adapt just as the application itself adapts.

There has been relatively little published work on tools to support program comprehension for SOA. Most of that work has concentrated on dynamic analysis tools to analyze patterns of execution from a running system instead of studying the artifacts that describe it. For example a group at IBM has developed a *Web Services Navigator* visualization tool that captures event logs from a running system and analyzes the resulting data to identify logic and performance problems (De Pauw et al. 2005). Another dynamic approach recovers a sequence diagram showing how a particular user feature executes. It does this by comparing inter-process messages collected when the feature is running with background messages collected when the system is performing other tasks (Coffey et al. 2010). For testing of an external service, another proposal is to start with a model of the expected sequence of interactions and generate test cases which are used to probe the running service and confirm or reject the model (Halle et al. 2010).

Dynamic analysis approaches to program comprehension have substantial advantages, especially in visualizing the complex, dynamically-changing interaction patterns of a SOA composite application. They do, however, require preparing tests and setting up a running copy of the application. The environment must support collecting and correlating traces or logs from multiple nodes. There is thus a

considerable amount of set-up work required, which may need to be repeated if the partners or technology of the application change.

## 4 Basic Search for SOA Evolution

Our research group has been focusing on search technology as a possibly simpler foundation for tools for SOA's open world. Text search has the advantage of being a well-established and well-known paradigm for gathering knowledge, as is evidenced by the overwhelming adoption of search services such as Google™ and Bing™. Search allows the efficient collection of information across a wide range of sources and document types, though the cost of this generality is that almost all the work of comprehension is put upon the user.

We have been experimenting for some time with a specialized search system for SOA evolution called *SOAMiner*. Our tool indexes a large collection of artifacts related to a particular SOA composite application and allows queries to locate information to support a particular maintenance task.

The original motivation for SOAMiner came from studying a small SOA composite application provided as a tutorial for a well-known open-source Integrated Development Environment (IDE). While intended to be a simple example, the whole application once deployed consisted of no less than 129 files distributed across 49 directories, not counting files actually deployed to the server! The most important artifacts were WSDL interface specification files backed up by XSDs for the industry-standard data types that were passed in the messages. The services were orchestrated by code in Business Process Execution Language (BPEL).

We found that it was very difficult to navigate the many interconnections between and within these artifacts and thus understand the overall structure of the composite application. Using the IDE helped somewhat, but relying on it would mean that any company owning the application would be dependent on that particular tool vendor. (As it happened, within a year and after a corporate take-over, the BPEL features of this particular IDE were no longer available.)

We noted that all three kinds of artifacts (WSDL, XSD, and BPEL) have XML structure and most of the information about interconnections was contained in attributes within the XML tags. Thus the first prototype of SOAMiner focused on extracting and searching text from such tags and was applicable to any file with XML structure[1]. We performed some initial studies using small datasets to let students evaluate the usability of the tool and larger ones to confirm acceptable performance (White et al. 2011).

[1] An online demonstration of this initial prototype may be viewed at http://soademo.cs.uwf.edu/SOASearch/

## 5 Knowledge-Enhanced Search for SOA Evolution

Our initial SOAMiner studies showed that the search approach was very powerful in locating information in the large corpus of artifacts of a SOA composite application. Yet it left the user with the often difficult task of understanding the results from each search query. The search simply displayed the XML tags that matched the query so the user had to supply the mental context of each match and often had to make multiple searches to trace through the relationships.

As an illustration, consider trying to understand service relationships in a simple system such as WebAutoParts.com, a SOA composite application that we have used in some of our case studies (Figure 2). WebAutoParts.com is a hypothetical online automobile parts supplier (Wilde et al. 2012).

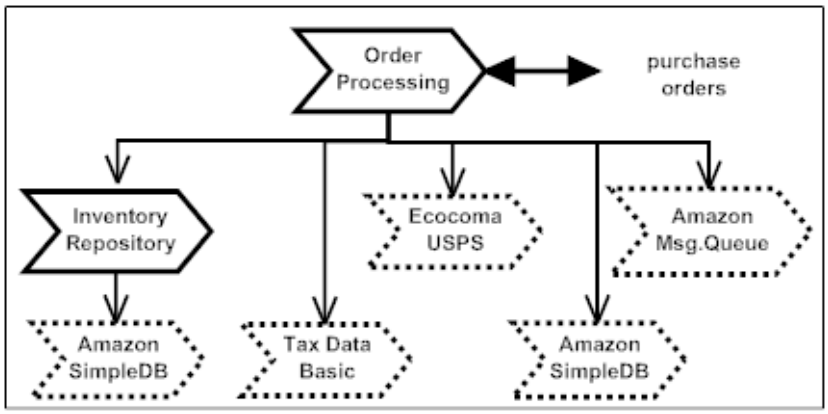


Fig. 2 - WebAutoParts Internal(solid) and External(dotted) Services

The WebAutoParts order processing workflow of Figure 2 has two stubbed in-house services in BPEL (Order Processing and Inventory Repository) and four external services represented by WSDLs and XSDs from three well-known vendors:

1. Amazon Web Services - Amazon Simple DB (database) and Message Queue (message queuing)
2. StrikeIron.com - Tax Data Basic (sales tax rates)
3. Ecocoma - USPS (shipping costs)

In this work flow, a purchase order is received, inventory availability is checked using the Inventory Repository service, American state sales tax is added depending on customer location, and USPS shipping costs are computed. The purchase order is then stored using Simple DB and a message is sent via the Message Queue service to trigger order fulfillment at the warehouse.

Suppose a Software Engineer unfamiliar with this application is trying to implement a change to the database design and needs to know what data is passed

when Order Processing checks inventory levels. If he has extensive BPEL/Web Services experience he might figure this out using a series of searches:

1. Search the Order Processing BPEL file to find the <invoke> tag that is checking inventory. That provides him a partnerLink. Then search the partnerLinks to get the partnerLinkType which turns out to be IRepositoryLinkType.
2. However the BPEL provides no indication of which service implements this link type, so the Software Engineer now searches all the WSDL documents for that link type. He will find it in InventoryRepositoryArtifacts.wsdl with a pointer to the WSDL portType for the service. The portType in turn gives the <operation> tag and its input and output message names. A further search on the message name reveals that the message contains an element called inventoryQuery.
3. However inventoryQuery is not defined within the WSDL so the Software Engineer now has to search XSDs to eventually locate the definition of inventoryQuery, determine its type, and from its type finally conclude what data fields are being passed.

This is, of course, just one example of the many relationships a Software Engineer may need to navigate to be able to understand a SOA composite application. The first prototype of SOAMiner greatly facilitates such searches, but not the process of establishing the relationships.

Clearly what is needed is some form of reverse engineering or abstraction to aid in comprehension. Where possible we would like to be able to search the artifacts making up a SOA application and then return abstractions that would quickly provide meaningful information to a maintainer. This raises two questions:

1. What are the important abstractions for SOA maintenance/evolution?
2. How can we provide abstractions while living with the changes of the SOA open world?

To identify important abstractions we conducted two case studies using the SOAMiner prototype. Both studies were informal; a small number of participants were asked to answer questions about a SOA system using the prototype while "thinking out loud" about their actions. They were observed while performing their task and then debriefed afterwards to capture their impressions and suggestions. The questions were chosen based on the kinds of search that Software Engineers have been found to use while developing and maintaining pre-SOA software systems (Sim et al. 1998).

The complete design and results of the case studies have been reported elsewhere (Reichherzer et al. 2011, White et al. 2012). However three abstractions stood out among the suggestions from study participants:

- **A compact representation of a service**: A WSDL file with its associated XSDs provides a representation of a service interface that is very difficult for humans to handle. In the great majority of cases the service could be displayed

as a simple tree showing the service, its operations, the input/output messages for each operation, and the data types for each message.

- **Compact data type summaries**: XML Schema Descriptions (XSDs) are used in WSDLs to describe the data passed in messages; the XSD tags may be incorporated directly into the WSDL or else imported from separate files. In both cases, the data description may be complex and dispersed with multiple levels of type and element descriptions which reference each other. For most cases a tree representation or an E-R diagram could be constructed that would be far easier for Software Engineers to grasp.
- **BPEL invoke relationships**: As shown by the WebAutoParts example given earlier, it may take a long sequence of steps to trace out what services are actually being invoked from a given BPEL file. It would be possible to automate this trace to draw a tree representation of the invoke relationships and provide a picture similar to Figure 2.

Our second question was how to compute these and other abstractions given the open and evolving nature of SOA composite applications. We needed some flexible and extensible way of defining abstractions over XML artifacts.

We are currently experimenting with an add-on to SOAMiner that uses a rule-based system to compute abstractions from a forest of XML-structured documents. Each abstraction type is specified as a set of rules that describe how a tree-representation of a specific abstraction may be derived from the original XML. For initial tests we have implemented rules for the three abstractions identified in the case studies. It should be easy to add or remove rules to allow the search tool to adapt to different technologies.

## 6 Current Directions for SOA Evolution Support

Our group is currently researching two additional approaches to enhance support for SOA evolution.

1. Searching code and documentation
2. Incorporating ontological information to improve system comprehension

Our initial prototype of SOAMiner focused on finding effective ways to search files with XML structure since they are most characteristic of SOA. WSDLs, XSDs, BPEL and many deployment and middleware configuration files all have an XML structure. A key decision was choosing the right granularity for indexing and results. SOAMiner treats each tag in the corpus as a separate entity, since it would not be very useful to respond to a query with an entire WSDL which may contain hundreds of tags.

An obvious way to increase the effectiveness of the search would be to also search source code and documentation. Here again the key question will be to es-

tablish the granularity for search. Documents may have no structure at all, or a very loose HTML or Microsoft Word structure. Code on the other hand is highly structured, but there are many different structures depending on the syntax of the specific programming language. Unfortunately SOA code exists in numerous languages and versions and it is impractical, at least for a research project, to provide a tool which will parse them all. As well, a tool that depends too strongly on specific syntax would not seem appropriate for the SOA open world since it may rapidly become obsolete as languages change.

Our research is experimenting with ways of deconstructing code and documentation into meaningful fragments that balance the need for tool generality against the increased power that can be obtained by taking advantage of input structure.

A second research direction is to provide ontological support for SOA program comprehension. An ontology represents the set of concepts in a particular domain, together with the relationships between those concepts. Our group is working to prepare a set of ontologies that represent those concepts that may be important for a Software Engineer performing maintenance on a specific SOA composite application.

The Open Group has released an ontology for Service-Oriented Architecture which is very useful for describing the overall structure of a composite application, its actors, its services, and its data (Open Group 2010). However since it is largely technology-neutral it does not go into many of the implementation details that a maintainer would need to deal with. Also, for any particular SOA composite it would be useful to have a domain ontology that describes the relationships in the real world within which the software is operating.

Ontological descriptions could be useful by themselves, for example to provide a shared vocabulary for human discussion of a system. They could also be useful to enhance the usefulness of search tools such as SOAMiner and its rules-based extension. Search could be enhanced by providing synonyms for search queries, by creating cluster abstractions of related software elements, and by prioritizing the display of search results based on semantic information.

## 7 Conclusions

In this paper we have argued that SOA has the potential to be a truly transformational technology as it opens up new opportunities for interoperability between software systems. But by their nature interoperable systems imply an environment of distributed ownership and control. Managers will be reluctant to trust such an environment unless they can be confident in their ability to respond to external changes with agile software evolution, but agile evolution requires rapid program comprehension of complex and heterogeneous systems in an open world with changing partners and changing technologies.

Our group's approach has been to focus on flexible, enhanceable search-based tools for program comprehension. The tool objective should be to be smart where possible, but useful everywhere, so that tool performance degrades gracefully as SOA technologies change. Thus for SOAMiner, the eventual goal would be to:

1. Provide abstraction-enhanced and ontology-enhanced search where it can
2. Provide useful text-based search everywhere else
3. Progressively allow more searches to be moved into the first category.

This is only one possible research direction out of many. Model Driven Architecture (MDA), for example, is an approach to SOA development that could have significant value for maintenance. With MDA a composite application is defined by models; code is either generated automatically or else the model may be directly interpreted at runtime (Salhofer and Stadlhofer 2012). The program comprehension burden may be reduced if only models need to be maintained and if they are substantially easier to understand than code. Of course the challenge may be to compatibilize the MDA models across a system with distributed ownership.

Ideally, definition of practical tools to support SOA evolution should be a collaboration between industry and researchers. The range of questions that SOA maintainers will face is still far from clear and an industry-university dialog could be most useful. We have been working with industrial contacts in the Security and Software Engineering Research Center (S2ERC) but a broadening of the conversation could help target research on the most important practical obstacles to agile change. Better tools and methods for SOA evolution could then be a major enabler for the interoperable systems of the future.

**Acknowledgments**
Work described in this paper was partially supported by the University of West Florida Foundation under the Nystul Eminent Scholar Endowment and by the Blue Cross Blue Shield Association and by Intelligent Information Technologies, both industrial affiliates of the Security and Software Engineering Research Center (www.serc.net).

**References**

Canfora G, Di Penta M (2007) New Frontiers of Reverse Engineering. Proc Future of Software Engineering 2007, pp. 326-341, doi:10.1109/FOSE.2007.15.

Coffey J, White L, Wilde N, Simmons S (2010) Locating Software Features in a SOA Composite Application. Proc. 2010 Eighth IEEE European Conference on Web Services, ECOWS'10, pp. 99-106, doi:10.1109/ECOWS.2010.28.

De Pauw W, Lei M, Pring E, Villard L, Arnold M, Morar JF (2005) 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

Gold N, Bennett K (2004) Program Comprehension for Web Services. International Conference on Program Comprehension, 2004, IEEE Computer Society, doi:10.1109/wpc.2004.1311057

Gold N, Knight C, Mohan A, Munro M (2004) Understanding Service-Oriented Software. IEEE Software 2004; Vol. 21 No. 2, pp. 71-77. doi:10.1109/MS.2004.1270766.

Halle S, Bultan T, Hughes G, Alkhalaf M, Villemaire R (2010) Runtime Verification of Web Service Interface Contracts. IEEE Computer, vol. 43, no. 3, 2010, pp. 59-66, doi:10.1109/mc.2010.76.

Janssen M, Charalabidis Y, Kuk G, Cresswell T (2011) E-government Interoperability, Infrastructure and Architecture: State-of-the-art and Challenges. Journal of Theoretical and Applied Electronic Commerce Research, Vol. 6, No. 1, April 2011, pp. i-vii, doi: 10.4067/S0718-18762011000100001

Josuttis NM (2007) SOA in practice: The art of distributed system design, O'Reilly, ISBN 0-596-52955-4

Kontogiannis K (2008) Challenges and opportunities related to the design, deployment and operation of Web Services. Proc Frontiers of Software Maintenance, 2008, pp.11-20. doi:10.1109/FOSM.2008.4659244.

Lewis GA, Smith DB (2008) Service-Oriented Architecture and its implications for software maintenance and evolution. Proc Frontiers of Software Maintenance, 2008, pp. 1-10. doi:10.1109/FOSM.2008.4659243

Luthria H, Rabhi FA (2012) Service-Oriented Architectures: Myth or Reality? IEEE Software, Vol. 29, No. 4, July-August 2012, pp. 46-52

McGregor S, Russ T, Wilde N, Gabes JP, Hutchinson W, Duhon D, Raza A (2012) Experiences Implementing Interoperable SOA in a Security-Conscious Environment. S2ERC-TR-307, Security and Software Engineering Research Center (S2ERC), http://www.serc.net, June 6, 2012. Also available at http://www.cs.uwf.edu/~wilde/publications/TecRpt307/ Accessed July 2012

Open Group (2010), Service-Oriented Architecture Ontology, ISBN 1931624887, 2010, https://collaboration.opengroup.org/projects/soa-ontology/?gpid=483, Accessed August 8, 2012

Reichherzer T, El-Sheikh E, Wilde N, White L, Coffey J, and Simmons S (2011) Towards intelligent search support for web services evolution: identifying the right abstractions. 13th IEEE International Symposium on Web Systems Evolution (WSE-2011), pp.53-58, 30 Sept. 2011, doi: 10.1109/WSE.2011.6081819.

Salhofer P, Stadlhofer B (2012) Semantic MDA for E-Government Service Development. 45th Hawaii International Conference on System Sciences, pp. 2189-2198, doi:10.1109/HICSS.2012.524.

Scholl HJ, Klischewski R (2007) E-Government Integration and Interoperability: Framing the Research Agenda. International Journal of Public Administration, Vol. 30, No. 8-9, pp. 889-920, 2007, doi:10.1080/01900690701402668

Sim SE, Clarke CLA, Holt RC (1998) Archetypal source code searches: a survey of software developers and maintainers. Proc. 6th International Workshop on Program Comprehension, 1998. IWPC '98, pp. 180-187, doi: 10.1109/WPC.1998.693351.

van den Heuvel WJ, Zimmermann O, Leymann F, Lago P, Schieferdecker I, Zdun U, Avgeriou P (2009) Software service engineering: Tenets and challenges. PESOS 2009, pp.26-33, 18-19 May 2009, doi: 10.1109/PESOS.2009.5068816.

White LJ, Reichherzer T, Coffey J, Wilde N, Simmons S (2011) Maintenance of service oriented architecture composite applications: static and dynamic support. J. Softw. Maint. Evol.: Res. Pract.. doi: 10.1002/smr.568.

White L, Wilde N, Reichherzer T, El-Sheikh E, Goehring G, Baskin A, Hartmann B, Manea M (2012) Understanding Interoperable Systems: Challenges for the Maintenance of SOA Applications. 45th Hawaii International Conference on System Sciences, pp. 2199-2206, 2012, doi:10.1109/HICSS.2012.614

Wilde N, Coffey J, Reichherzer T, White L (2012) Open SOALab: Case Study Artifacts for SOA Research and Education. Principles of Engineering Service-Oriented Systems, PESOS 2012, Zurich, Switzerland, pp. 59-60, June 4, 2012, doi: 10.1109/PESOS.2012.6225941