Enhanced Search: An Approach to the Maintenance of Services Oriented Architectures

Norman Wilde, Douglas Leal, George Goehring, Christopher Terry
Department of Computer Science
University of West Florida
Pensacola, FL, USA
e-mail: nwilde@uwf.edu, douglas.leal@gmail.com, pensacoder@gmail.com, cterry@students.uwf.edu

***Abstract*— This paper describes the use of search techniques to ease the burden of software maintenance for Services Oriented Architecture composite applications. Services Oriented Architecture is a paradigm that offers many potential business and social benefits, especially because it creates opportunities for composite software applications that share data and functionality across organizational boundaries. However, along with these benefits will come new challenges in the maintenance of these applications. The first necessity in any software maintenance task is to comprehend how the existing software functions. To gain this comprehension, maintainers will need to study a bewildering variety of artifacts, ranging from XML-based interface descriptions, through source code in a variety of languages, to traditional text documents in many different formats. For some years, we have been experimenting with the use of modern search techniques, enhanced where possible by rule-based reasoning, to aid maintainers of composite applications in gathering the information they will need to do their jobs. In this paper, we describe version 2 of our SOAMiner search system and discuss how its design emerged from our experiences. While SOAMiner is still a prototype, we argue that search, enhanced and specialized for Services Oriented Architecture can provide useful support to maintainers of these very heterogeneous applications.**

Keywords-Services Oriented Architecture; SOA; Software Maintenance; Search; Rule-Based Systems.

I. INTRODUCTION

The last decade has seen the emergence of a new paradigm for large scale software applications often called Services Oriented Architecture (SOA). While definitions of SOA vary, the term usually refers to large *composite applications* implemented as large-grained services running on different nodes and communicating by message passing (see Figure 1). Implementation technologies differ, but often follow the Web Services interoperability standards.

The SOA architectural style has great potential to achieve business or social goals through interoperability across organizational boundaries. As an example of SOA, consider the CONNECT project, which provides a set of software and standard interfaces for health information exchanges in the United States [1]. The goal of CONNECT is to enable health data to follow a patient wherever he may need treatment.

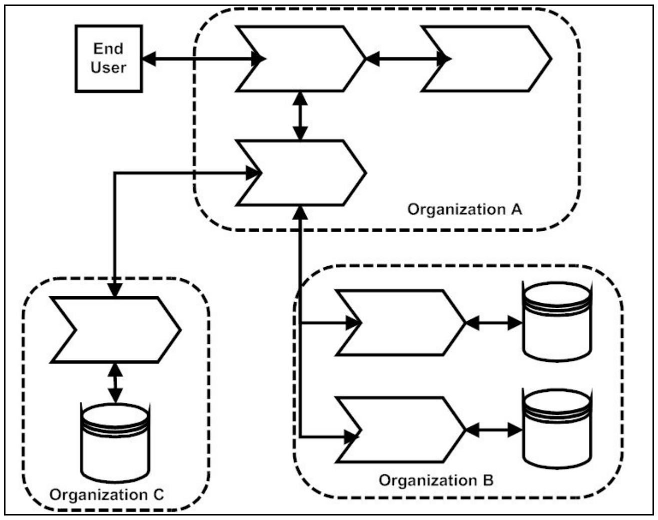

Figure 1. A SOA composite application with services from three partner organizations exchanging messages.

However, to achieve such benefits over the long term, SOA composite applications will have to be maintainable in a rapidly changing world. Several authors have pointed out characteristics of SOA that may make maintenance difficult [2][3][4]. Often, one such characteristic is distributed ownership, so that different services in the composite application are operated and maintained by different partner organizations. Thus, changes to specifications may need to be negotiated, coordination of updates may be complicated and the maintainer's information about some services may be incomplete. The mix of partners may change unpredictably over the application's lifetime, requiring quick re-engineering to adapt as services are offered or withdrawn. Critical security issues may emerge without warning, and it may be difficult to identify their impacts without knowing how partner services are implemented.

A traditional stumbling block in all software maintenance has been the need for program comprehension. The first question a maintainer must always ask is "how does the software work now?" Changes made to a software system without deep understanding can be highly error prone. A particular maintainer's problem has always been the *delocalized software plan* in which the original programmer's strategy for addressing some specific issue has been implemented by related code in several distant program modules [5]. Subtle faults may be introduced if a maintainer makes changes in one of these modules in ignorance of possible effects in others.

In SOA, the delocalization is not confined to a single executable but may spread across different services which, as we have seen, may have different owners. While every service has a published interface, which is sufficient to invoke it, in practice there are often additional data and operation sequencing constraints that must be learned by experience or by study of documentation [6].

There has been a modest amount of recent research on maintaining SOA applications. Papazoglou, Andrikopoulos, and Benbernou categorize changes into "deep" and "shallow" and discuss how to keep services compatible [7]. Several authors have proposed dynamic analysis approaches that analyze inter-process messages to pull together a view of execution across the multiple services. An early tool of this kind was IBM's Web Services Navigator, which provides several visualizations of message logs [8]. A later paper from the same group describes a process that looks more deeply into message contents to identify data correlations between different messages [9]. Yousefi and Sartipi propose analyzing dynamic call trees from distributed execution traces to identify features in a SOA application [10]. A different reverse engineering approach, which does not rely on executing the system, recovers concept maps from the interface descriptions as a starting point for knowledge engineering interviews with system experts [11].

Looking for a simpler and more flexible approach, for some time our group has been researching ways to exploit the power of modern search techniques and adapt them to the specific needs of SOA maintainers. The overall project is called SOAMiner and has gone through a series of prototyping and exploration phases [12]. In this paper, we will describe version 2.0 of SOAMiner, which incorporates the experience from these earlier studies. SOAMiner is built on top of the Apache Solr™ open-source search platform [13]. The new version of SOAMiner provides a combination of conventional text search, specialized search that exploits the structure of many SOA artifacts, and rule-base abstraction to provide summarized descriptions of SOA services and data.

In the next section of this paper, we explain how these three strategies emerged from our experience in applying search to SOA. Then, in Section III, we illustrate their application by showing how SOAMiner can address a maintenance scenario for a simple SOA composite application. Finally, in Section IV, we conclude with some thoughts about SOA and the evolution of SOA systems.

II. SEARCHING SOA ARTIFACTS

In trying to comprehend a SOA composite application, a maintainer must deal with a bewildering variety of artifacts. These may include XML documents that describe service interfaces, source code for service implementations, and any conventional documentation that a service provider has chosen to offer. In developing a search strategy for these different classes of artifacts a key decision is the granularity of response. If a search returns just the few words that match the query, then the maintainer will struggle to understand how these fit into the application as a whole. If a large volume of surrounding text is also returned, then the maintainer may be buried in extraneous details. In this section, we discuss our experiences in searching these different classes of SOA artifacts and the granularity we have chosen for each class.

A. Searching XML Artifacts

When SOA is implemented using Web Services, then much of the information about each service is coded in XML format as specified in one or more of the Web Services Standards ([14], Chapter 16). The most common standards cover Web Services Description Language (WSDL) to specify how to call a service and XML Schema Definitions (XSD) to specify the data exchanged in messages. Some SOA systems also use Business Process Execution Language (BPEL) which is essentially a programming language encoded in XML for orchestrating interactions among services.

The XML artifacts may often be very large; we have seen extreme WSDL's of over 1 MB and several thousand lines is not uncommon for an XSD. Such files are often generated by some tool but it may still be necessary for the maintainer to study them himself when trying to comprehend a service. The structure of these files does not facilitate human navigation.

For example, to identify the data types being used by a particular service a maintainer needs to read its WSDL "bottom up", starting from a <service> tag near the end, locating the <port> tag it contains, navigating from there to a referenced <binding> tag, which in turn references the <portType>. From there the <portType> encloses a set of <operations> with <input> and <output> tags each pointing to a <message> tag. However, the maintainer is still not finished because in most cases each <message> simply references the actual data types, which are either enclosed within the <types> section near the beginning of the WSDL, or possibly contained within a completely separate XSD file [14].

Generic search approaches, such as a text editor's 'find' or a document-oriented web search engine, do not work very well on these XML artifacts. Such approaches ignore too much context because they are unaware of the significance of XML tag names and of the information conveyed by element nesting. Figure 2 provides one example showing how a port type is defined in a WSDL. Element nesting determines that the messages relate to the operation and the operation to the port type.

```
<!-- portType for  the InventoryRepository process -->
<portType name="InventoryRepositoryPortType">
  <operation name="checkInventory">
    <input  message="tns:InventoryRepositoryRequestMessage" />
    <output message="tns:InventoryRepositoryResponseMessage"/>
  </operation>
</portType>
```

Figure 2. Portion of a WSDL showing the definition of an operation within a port type.

When SOAMiner searches XML, the basic granularity is the element start tag, so that a search for "checkInventory" would return just the <operation> tag from Figure 2. If the system is large, the user can specify a faceted search to limit the results to a single tag type. As well, in SOAMiner we also attach to each tag its parent and any children in the XML document. Thus, if using our search GUI, the user could hover over that result and see the surrounding <portType> and the <input> and <output> tags. That provides the maintainer with a few more hints as to the context of each search result so he can focus quickly on the results that are of most interest.

B. Rule-Based Abstraction from XML Artifacts

However, we can do even better than that by exploiting knowledge about the semantics of the different XML tags through a process of rule-based abstraction. We have implemented such abstractions in a component of SOAMiner called SOAIntel. An expert can specify a set of rules for SOAIntel to define an abstraction, which summarizes some characteristic of a class of SOA implementations. For example, the rules could describe the above mentioned chain of reasoning to relate the service to the data items in its input and output messages. The resulting abstraction would be a compact description of the service, its operations, and their messages.

Rules are encoded using the DROOLS Expert rule-based system [15]. SOAIntel uses the rules and the DROOLS reasoning engine to analyze the XML inputs and produce a set of abstractions. These abstractions are then loaded into the SOAMiner index so that they may also be returned by SOAMiner searches. Thus, a maintainer searching on "checkInventory" would also find that this operation is part of a service abstraction named InventoryRepository and thus see that its messages use an element called inventoryQuery, etc.

The rule-based abstraction process is very flexible, so that new rule sets can easily be added to cope with changes to the Web Services standards or with specific maintenance needs for any particular class of composite applications [16].

C. Searching Source Code and Documentation

The source code for a SOA service may be in any of a multitude of languages; in fact one of the objectives of SOA is to allow services written in one language to invoke transparently services written in another. In several of the most common languages, such as Java and C#, much of the code is commonly generated within an Integrated Development Environment (IDE). For example, a Java developer using NetBeans will call a tool called *wsimport* to read a WSDL interface description and generate Java classes for the message data types and a shell service implementation. The generated code can be rather obscure, and as well makes use of many Java annotations to guide the run-time environment as the service executes. While the availability of generated code greatly reduces the amount of code a service developer needs to program, it also creates complex mechanisms that a maintainer may need to learn.

The diversity of source languages and run-time mechanisms makes it very difficult to develop a general code search tool with any intelligence. Instead, for now, SOAMiner falls back on normal text search, to locate lines of code matching a given query string.

The situation is similar for natural language documentation, which may be in text, Portable Document Format (PDF), HyperText Markup Language (HTML) or some word processor format. In the future, it may be possible to apply text mining techniques such as text classification and text clustering to these documents, but for now SOAMiner relies on general text search, using the facilities of Apache Tika™ to parse each document format and extract the text contents [17]. Since lines and even paragraphing may not be meaningful for all document types, each SOAMiner search simply returns the entire document contents.

D. Search semantics for SOA

Software Engineers search software for many reasons, but two very common ones are *concept location* [18] and *impact analysis* [19]. Concept location has to do with finding the places in a software system where some particular concept is addressed. For example, one could ask "where are font changes handled in this word processor?" and search for the concept "font" in code, documentation, etc. On the other hand, impact analysis is concerned with establishing the scope of a needed change. If, for example, the Software Engineer has determined that a particular function needs to be modified, then he needs to look at all the places that function is called so that he can see what the change may impact.

One of the observations we made after working with the first versions of SOAMiner was that the semantics of these two kinds of search are really quite different. Concept location will usually use natural language semantics and most of the techniques used in search engines should be applied. For example, queries should be stemmed and case insensitive, so that "font" will match "fonts" or "Font". Query words should break on punctuation or case changes so that again "font" will match "font_change" or "fontChange".

However, for impact analysis the rules should be very different and use identifier semantics. Normally the Software Engineer will have located a particular variable or function name, such as "fontChange", and only wants to locate occurrences of that identifier. A search with natural language semantics would return all text where either "font" or "change" appeared, and that would be far too many places to examine.

The solution adopted in version 2 of SOAMiner is to provide two alternate indexes, one using natural language semantics and the other using more restrictive identifier semantics. The user may choose which to use for any particular query depending on the results sought.

III. ILLUSTRATIVE EXAMPLE

To provide an example of the power of enhanced search, we may apply it to a simple SOA composite application called WebAutoParts.com, a hypothetical on-line automobile parts dealer. The owners of WebAutoParts have adopted an agile development strategy, in which a small amount of internal code orchestrates commercially available services to provide needed functionality quickly [20]. WebAutoParts is an academic system, not a real application, so several of its components are stubs instead of full code. Still it models the complexity of a real application since it consists of in-house services with BPEL and some other code artifacts, WSDL artifacts that describe external services from well know vendors (e.g., Amazon Web Services, StrikeIron.com), and XSD schemas to define data types used in system messages (see Table I). The application provides an order processing work flow (see Figure 3) in which incoming orders are first checked to confirm that inventory is available, then sales tax and shipping are computed, and finally the order is stored and a note placed in a message queue to trigger order fulfillment (packing and shipping).

To illustrate the use of SOAIntel, two rule sets were written that generate two different kinds of abstractions from the XML files. The first abstraction is a compact service summary that shows the service and port type, the operations in that port type, and the names of the input and output messages of each operation. For the great majority of services this summary will obviate the need to step through the WSDL tag by tag to understand the service interface.

The second rule set generates a data type summary abstraction that shows the different data items making up a message. During our earlier studies with the first version of SOAMiner users requested this kind of summary to help them navigate the complexities of data typing in SOA [21]. The Web Services standards give developers a wide variety of ways to define the data in messages, and the definitions may look very different even if the final message content is much the same. For example, the data definitions may be in different places, either in the <types> section of the WSDL itself, or else located in an associated XSD file.

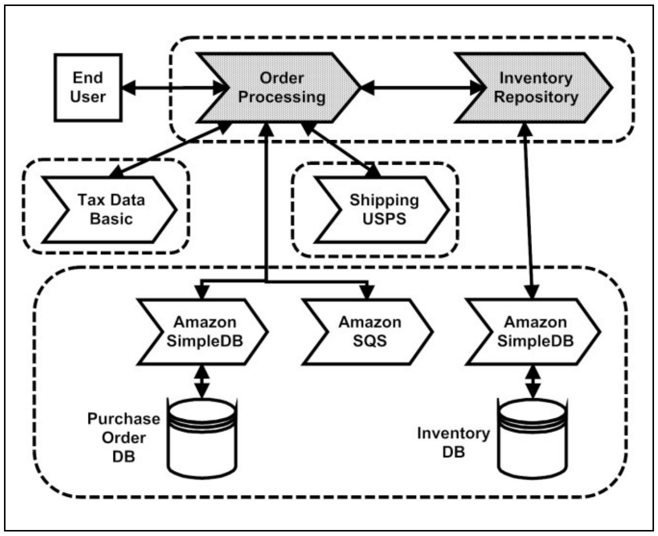

Figure 3. The WebAutoParts order processing work flow showing internal (shaded) and external services.

TABLE I. WEBAUTOPARTS ARTIFACTS

File Type	Files	Lines
WSDL (XML)	6	2433
BPEL (XML)	2	189
XSD (XML)	2	64
JAVA (Code)	6	450
C# (Code)	3	336
Microsoft Word	1	718
HTML	1	374
PDF	1	230

The style of the definition can also vary widely since developers may use different combinations of XSD elements, references and complex types to say much the same thing. (The different design patterns have been given names such as *Russian Doll* and *Venetian Blind*, and each has its own advantages and drawbacks in terms of re-usability and visibility [22]). To reduce this confusion our second rule set extracts a simple list of the data items making up each message, independent of the location or form of the definition.

To see how a maintainer could use enhanced search in studying an application such as WebAutoParts, consider the following hypothetical scenario. Employees of WebAutoParts have reported that, occasionally when packing and shipping an order, an item is found to be out of stock, even though the order processing workflow showed that inventory was available. Something in the computation of stock levels is obviously in error. The problem is passed to a software engineer for action. Let us suppose that this software engineer has little previous experience with the order processing work flow of Figure 3.

Table I enumerates the artifacts that describe WebAutoParts. There are a total of 10 XML files, 9 code files and 3 documentation files. These are loaded into the SOAMiner Solr index using the parsers for XML, code and documentation respectively.

As always, the software engineer's first question is "How are stock levels computed now?" He uses SOAMiner to do a concept location query on "stock". The results are shown in column A of Table II. Just one documentation file was located and he picks that as the starting point most likely to give him an overview of the situation. The documentation file turns out to provide a general description of order processing and provides roughly the same information that readers of this paper have already seen. While it mentions briefly that stock levels are checked it does not say how. It does, however, show the overall workflow and indicates what services participate in it.

TABLE II. RESULTS OF QUERIES ON WEBAUTOPARTS

	Column A concept location "stock"	**Column B impact analysis "numberInStock"**
XML tags	2 element tags	2 element tags
Abstractions	2 message data items abstractions	2 message data items abstractions
Code lines	13 Java, 6 C#	9 Java, 3 C#
Documentation files	1 Word doc	none

The software engineer next looks at the two abstractions, which show the data items making up the InventoryRepositoryRequestMessage and the InventoryRepositoryResponseMessage. He can see immediately that these are respectively the input and output messages of an operation called checkInventory in the InventoryRepository service. His query on "stock" matched a data item named "numberInStock" which is contained in both messages. (Concept location queries use the natural language semantics index in which words break on changes of case, so the query word "stock" matches "numberInStock".)

It seems highly likely that the error involves in some way the numberInStock data item and the checkInventory operation. Thus next the software engineer does an impact analysis query on "numberInStock". The query uses the identifier semantics index so it will only find exact matches to that string. The results are shown in Column B of Table II. The query finds the same two XML tags and message data items abstractions, but it locates a smaller set of code lines, reducing the places the software engineer needs to look. The code lines are in a Java implementation of the InventoryRepository service and a shell C# implementation of a test client to that service.

Now that he has the big picture, the software engineer can start looking at code. Here he can make use of specialized IDE's for Java or C# having their own very good search facilities. Combining his overall view of the workflow with a little analysis reveals a classic "omitted logic" problem [23]; while InventoryRepository gets the correct value for the numberInStock at any moment, there is nothing to prevent a second order from checking that same stock level before the first has completed order fulfillment, so the same stock may be committed twice. As often occurs, the error is not really "in" any particular service, but is a consequence of implicit assumptions made as the different services were orchestrated together.

IV. CONCLUSIONS

In this paper, we have argued that Services Oriented Architectures will not attain their full potential unless these applications can be rapidly maintained. SOA applications will need to provide high availability in a world with changing requirements, shifting partner alliances and emergent security threats. Their maintainers will need to gather information quickly to comprehend and respond correctly to each challenge.

In confronting these challenges, maintainers will need both good governance and good tools. In SOA, the term "governance" refers to the set of policies, rules, and enforcement mechanisms for developing, using, and evolving SOA-based systems [24]. There is a great danger of organizations trying to go too far too fast with SOA and creating composite applications that go beyond the organization's capacity to maintain. The scope of applications, the range of implementation technologies and the rate of requirements creep need to be limited to match organizational capabilities.

If the organization provides a reasonable governance framework, then well qualified software engineers with good tools should be able to do the job. Our SOAMiner is only intended as one example of the sorts of tools that will be needed. The current version remains a prototype. There are some places where it is less precise than we would wish, for example in the handling of namespaces. The user interface remains a work in progress. However, we feel that the flexibility provided by the combination of modern search with rule-based abstraction is well suited to the changing world of SOA. The search techniques can be applied to just about any kind of artifact encountered in a SOA system, while the abstraction mechanism can leverage a rule base that grows with experience. Thus, a search tool like SOAMiner can provide some useful information almost all the time, and can provide better and better information as experience grows.

It will be interesting to see how well the SOAMiner approach scales to real-world SOA. Limited experience with one larger system indicated that the pure search aspects provided excellent performance, which was to be expected since the Solr search engine was developed with large data sets in mind. The scalability of the rule-based abstractions may be more problematic. Our limited experience so far is that performance can depend on how well the rules are crafted to exploit the DROOLS index structure.

The evolution of SOA systems will never be easy, but with thoughtful governance, skilled software engineers and good tools, it should be possible to manage the challenges.

ACKNOWLEDGMENT

Work described in this paper was partially supported by the University of West Florida Foundation under the Nystul Eminent Scholar Endowment. Apache, Apache Solr, and Apache Tika are trademarks of The Apache Software Foundation. Used with permission. No endorsement by The Apache Software Foundation is implied by the use of these marks.

REFERENCES

[1] "What is CONNECT?", Internet: http://www.connectopensource.org/about/what-is-connect, link accessed 2014.07.22.

[2] N. Gold and K. Bennett, "Program comprehension for web services", International Workshop on Program Comprehension (IWPC'04), June 2004, pp. 151-160, doi: 10.1109/wpc.2004.1311057.

[3] N. Gold, C. Knight, A. Mohan, and M. Munro, "Understanding service-oriented software", IEEE Software, Vol. 21, March 2004, pp. 71-77, doi: 10.1109/ms.2004.1270766.

[4] G. Lewis and D. Smith, "Service-Oriented Architecture and its implications for software maintenance and evolution", Frontiers of Software Maintenance, FoSM 2008, Sept. 2008, pp. 1-10, doi: 10.1109/fosm.2008.4659243.

[5] S. Letovsky and E. Soloway, "Delocalized plans and program comprehension", IEEE Software, vol.3, no.3, May 1986, pp. 41-49, doi: 10.1109/MS.1986.233414.

[6] S. Halle, T. Bultan, G. Hughes, M. Alkhalaf, and R. Villemaire, "Runtime verification of web service interface

contracts", IEEE Computer, Vol. 43, March 2010, pp. 59-66, doi: 10.1109/mc.2010.76.

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

[8] W. De Pauw, et al., "Web services navigator: visualizing the execution of web services", IBM Systems Journal, Vol. 44, No. 4, Oct. 2005, pp. 821-845, doi: 10.1147/sj.444.0821.

[9] W. De Pauw, R. Hoch, and Y. Huang, "Discovering Conversations in Web Services Using Semantic Correlation Analysis", IEEE 20th International Conference on Web Services, ICWS'2007, July 2007, pp. 639-646, doi: 10.1109/ICWS.2007.200.

[10] A. Yousefi and K. Sartipi, "Identifying distributed features in SOA by mining dynamic call trees", IEEE International Conference on Software Maintenance (ICSM), Sept. 2011, pp. 73-82, doi: 10.1109/ICSM.2011.6080774.

[11] J. Coffey, T. Reichherzer, B. Owsnick-Klewe, and N. Wilde, "Automated Concept Map Generation from Service-Oriented Architecture Artifacts", Proc. of the Fifth Int. Conference on Concept Mapping CMC2012, Sept. 2012, pp. 49-56.

[12] E. El-Sheikh, et al., "Towards enhanced program comprehension for service oriented architecture (SOA) Systems", Journal of Software Engineering and Applications, Vol. 6, No. 9, Sept. 2013, pp. 435-445, doi: 10.4236/jsea.2013.69054.

[13] "Apache Lucine, Apache Solr", Internet: https://lucene.apache.org/solr/, link accessed 2014.07.22.

[14] N. Josuttis, SOA in Practice: The Art of Distributed System Design, O'Reilly, 2007, ISBN: 0-596-52955-4.

[15] "Drools - JBoss Community", Internet: http://drools.jboss.org/, link accessed 2014.07.22.

[16] G. Goehring, et al., "A knowledge-based system approach for extracting abstractions from service oriented architecture artifacts", International Journal of Advanced Research in Artificial Intelligence (IJARAI), Vol. 2, No.3, 2013, pp. 45-52, doi: 10.14569/IJARAI.2013.020307.

[17] "Apache Tika", Internet: http://tika.apache.org/, link accessed 2014.07.22.

[18] V. Rajlich and N. Wilde, "The role of concepts in program comprehension", 10th International Workshop on Program Comprehension, June 2002, pp. 271-278, doi: 10.1109/WPC.2002.1021348.

[19] M. Petrenko and V. Rajlich, "Variable granularity for improving precision of impact analysis", International Conference on Program Comprehension, 2009. ICPC'09, May 2009, pp. 10-19, doi: 10.1109/ICPC.2009.5090023.

[20] N. Wilde, J. Coffey, T. Reichherzer, and L. White, "Open SOALab: Case study artifacts for SOA research and education", Principles of Engineering Service-Oriented Systems, PESOS 2012, June 2012, pp. 59-60, doi: 10.1109/PESOS.2012.6225941.

[21] L. White, et al., "Understanding interoperable systems: Challenges for the maintenance of SOA applications", 45th Hawaii International Conference on System Sciences (HICSS), January 2012, pp. 2199-2206, doi: 10.1109/HICSS.2012.614.

[22] E. Hewitt, Java SOA Cookbook, O'Reilly, 2009, ISBN: 978-0-596-52072-4.

[23] R. L. Glass, "Persistent software errors", IEEE Transactions on Software Engineering, Vol. SE-2, No. 2, March 1981, pp 162-168, doi: 10.1109/TSE.1981.230831.

[24] G. Lewis and D. Smith, "Four pillars of service-oriented architecture", CrossTalk, September 2007, pp. 10-13. Available from: http://www.crosstalkonline.org/storage/issue-archives/2007/200709/200709-Lewis.pdf, link accessed 2014.07.22.