Locating Software Features in a SOA Composite Application

John Coffey\textsuperscript{1}, Laura White\textsuperscript{2}, Sharon Simmons\textsuperscript{3}, Norman Wilde\textsuperscript{4}
\{\textsuperscript{1}jcoffey|\textsuperscript{2}lwhite|\textsuperscript{4}nwilde\}@uwf.edu  \textsuperscript{3}simmonsj@jmu.edu

Executive Summary

This report contains a description of the use of a teaching and research program suite named OpenSOALab to perform feature location in a composite application. In this context, a software feature refers to software components that provide specific functionality. The composite application encompasses a system in which hotel brokers identify rooms meeting various criteria from among several hotel chains in multiple countries, and then exchanges the necessary amount of currency, using a currency broker to get several quotes and select the best one. The currency broker in turn uses two services: an authentication service and a settlement house. The various service interfaces are exposed via WSDLs. The system, running on Apache with php and nuSOAP, uses Apache’s forensic log module and micro-second time stamps to generate data that is input into a Feature Sequence Viewer (FSV). The FSV produces a browsable graphical representation of the messages in the system. The FSV employs a component relevance index (p_c) that can be adjusted by the user to separate and display the messages that are most likely to be in the feature of interest from those more likely not to be. Three experiments of increasing complexity were performed to demonstrate the ability of this approach to extract feature messages from irrelevant messages.

This report may be cited as S\textsuperscript{2}ERC-TR-304, Security and Software Research Center (S\textsuperscript{2}ERC), http://www.serc.net, July 6, 2010.

Table of Contents

1. Introduction .................................................. 2
2. Program Understanding in Composite Applications .................................................. 2
3. The Current Study .................................................. 3
   3.1 OpenSOALab and Feature Sequence Viewer .................................................. 3
   3.2 Descriptions of the Experiments .................................................. 5
      3.2.1 Experiment 1 .................................................. 5
      3.2.2 Experiment 2 .................................................. 6
      3.2.3 Experiment 3 .................................................. 6
4. Methods .................................................. 6
5. Results .................................................. 7
   5.1 Experiments 1 and 2 .................................................. 7
   5.2 Experiment 3 .................................................. 7
6. Acknowledgments .................................................. 9
7. Conclusions .................................................. 9
8. References .................................................. 10

1. Introduction

The term feature with regard to computer software is taken to mean different things by different groups. For instance, the IEEE defines the term feature quite broadly as "A distinguishing characteristic of a software item (e.g., performance, portability, or functionality)." \cite{1} Within the program understanding community, a software feature is taken to be a specific functionality provided by the software. The feature location problem is the problem of locating the program components that provide the feature. By this definition, a feature might be relatively fine-grained or course-grained, and the associated code might likewise be viewed broadly (at the module or class level) or narrowly (at the statement level).
The problem of locating software features varies depending on the system under consideration. For instance, in large, single-threaded applications, the problem may be one of identifying small sections of code somewhere in a very large program, developed by many people over a long period of time, on the basis of some sort of static or dynamic analysis. In distributed systems (defined as those executed across multiple physical machines with true concurrent execution) the problem of feature location might be couched in terms of finding code components that are executing in the same time interval [2].

The problem of feature location in composite applications based upon Web services is different yet again. First, execution of a composite application might be centrally controlled (orchestrated), or it might execute as a peer-to-peer process (choreographed). Large composite systems will most typically execute according to a script developed in a business process modeling language such as BPEL or BPMN, which are quite expensive to start up and maintain [3]. The BPEL file provides a course-grained view of features that execute. Other smaller heterogeneous systems are assembled in a more ad-hoc fashion and have very different software understanding issues. Such composite applications might be hand-built out of service interfaces exposed through WSDLs with no global view of the choreography involved. Maintenance of such systems certainly might require a more fine-grained view of their features. The focus in this paper will be on this category of system.

The remainder of this article contains a description of literature and prior work on program understanding for composite applications. It contains a description of three experiments pertaining to feature location that were performed using OpenSOALab [4] and Feature Sequence Viewer (FSV) software. OpenSOALab is a teaching tool and research testbed that provides and consumes an integrated suite of web services pertaining to creation and viewing of hotel reservations and currency exchange. FSV is software that provides a graphical representation of the sequence of messages that indicate executing features. FSV implements an algorithm that is able to make probabilistic differentiations between so-called marker components that are likely to be in a feature of interest and irrelevant components that are not in the feature. The experiments, which involve feature location in the context of increasingly complex system interactions, were performed in isolation and in the presence of generated noise messages. This report concludes with descriptions of the experiments, several graphics created by the FSV, and conclusions that were drawn from the studies.

2. Program Understanding in Composite Applications

Traditional approaches to software understanding present a context for program understanding in composite applications, and will be briefly discussed in this section. Additionally, literature pertaining to composite event detection through logging and analyzing messages, performance modeling to judge QoS, and for service error detection and recovery will be reviewed.

Broadly speaking, feature location may be fostered either by static or dynamic analyses. Program slicing [5] is a static technique that identifies all code involved in the computation of a variable of interest. Other techniques range from basic text searches such as those afforded by grep [6] to analysis of a program’s dependency or call graph [7]. Dynamic approaches to program understanding include the technique of software reconnaissance [8][9]. This approach to feature location might involve running a system with several test cases that both include and exclude the feature of interest, and then taking the set difference between the set of components that executed in the feature and those that didn’t, or computation of a component relevance index (p_c) that represents the likelihood that a component is an element of a feature of interest.

Composite event detection has the goal of identifying patterns of events through system messages. Pientuch, Shand, and Bacon [10] describe a composite event detection service developed in Java. Their system provides event detection on top of middleware using JMS – Java Messaging Service. Representations of features are made in finite state automata indicating state changes in the executing software. Chan [11] suggests that the need exists for improved runtime monitoring and recovery in composite applications. He compares and contrasts his approach with three others. This approach might be characterized as dynamic monitoring of BPEL-governed processes. A separate approach suggests extensions to BPEL engines for monitoring/recovery. Emphasis is on dynamically detecting error conditions and fixing them on the fly. Chan’s article contains a description of Dynamo [12], which provides intrusive monitoring for BPEL processes and error detection/correction. VieDAME [13] is a less-intrusive system that monitors a BPEL process to determine if it meets quality of service specifications. Chan also mentions work by Subramanian, Thiran, Narendra, Mostefaoui, and Maamar [14] in which a proposal to extend the BPEL engine and standard itself for monitoring and recovery is made. Broadly speaking, this category of paper seeks to explore alternative service architectures to improve performance. This category of paper proceeds from the assumption that the services work. D’Ambrogio [15] talks about P-WSDL performance enabled WSDL. Brebner [16] describes a tool to provide performance modeling for SOA, again with the goal of meeting specified service level agreements. Narendra and Orriens [17] discuss rule-driven modeling of composite Web services in terms of functional and non-functional requirements. The couch their monitoring scheme in terms of “who,” “what,” “when,” “where,” and “why.”
While testing tools are often more focused on verification of software than on understanding it, they indirectly provide support for program comprehension. There has been a fair amount of work on testing tools for the SOA environment and many, such as the PLASTIC environment, include system monitoring that might be used to provide support for program comprehension [17]. Sneed describes a SOA testing tool and mentions the potential value of tracing the path of a web service request through the system [18]. The work described in this report is one approach to that goal. Finally, Halle et al. point out that may parts of a service are not covered in the WSDL: there may be additional preconditions, data constraints, and control flow constraints [19]. They do runtime testing to aid in service comprehension. They hypothesize what the complete service contract should be and check it by automatically generating multiple sequences of invocations against the live service.

Errors can occur in many places within composite applications. A partial list includes: in adapters written to enable legacy systems as Web services, in WSDL specifications, and in undetected errors in the business logic of the services themselves. Either a course-grained or fine-grained approach to error detection may be used. The BPEL specification allows for two faults that can be thrown [20]. An invalid-request indicates that the request message has either invalid information, is missing required information, or includes requests for functionality that is not supported. A processing-error indicates that the operation failed. Return information is fairly terse; for instance, if the request fails to identify a proper remote function, the response does not contain a process definition.

3. The Current Study

In the current study we imagine a software engineer tasked with maintaining a SOA application. This person is trying to understand how a particular feature works, either because a bug has been reported or because an enhancement is needed. As is common in industry, he has a multi-computer laboratory available to him for testing the SOA application, but other engineers performing other tasks share the laboratory. There are thus two cases:

1. He may be able to find a moment when the different servers in the laboratory are free of other use, so that he may execute his tests and check logs in the knowledge that nothing else is going on, or

2. He has to work at the same time as others, so that there is always some "noise" activity that will show up in traces and logs. He must somehow distinguish the effect of his tests from this noise.

While case (1) is obviously ideal, we understand that case (2) may be much more probable. We have developed the Feature Sequence Viewer (FSV) tool to aid feature comprehension in this second, "noisy" environment. FSV, as its name indicates, attempts to reconstruct a sequence diagram showing the messages that implement the feature. Sequence diagrams are a common design representation and have the advantage of being very familiar to most software engineers. They have been a common goal of reverse engineering efforts, though they have been used more commonly for showing method invocations in object oriented programs, rather than for understanding inter-process messages as in SOA [21]. The system was used under both normal operating conditions and with a “gap in the services” scenario in which the software behaved as expected, but one service in the composite application was not available. Experiments were also run with features executing by themselves, and in the presence of noise – irrelevant transactions that occurred before, during and after the feature of interest. The following sections contain descriptions of OpenSOALab and FSV, the design and conduct of the experiments, and results.

3.1. OpenSOALab and Feature Sequence Viewer

Our tests were performed using SOA composite applications from Open SOALab. Open SOALab is a teaching and research testbed containing services developed by different groups of students and faculty over several years. The services are intended to model, at a small scale, real-world SOA services developed and deployed by different providers. OpenSOALab is comprised of two major components, a currency exchange system and a hotel system. The currency exchange system exposes an interface with a total of 10 remotely invoked procedures, and the hotel system has 7 remote procedures. Services are specified with WSDLs and invoked with SOAP over http. OpenSOALab runs on one or more Apache servers using nuSOAP and PHP. The Apache forensic log module was enabled for the system, and software included in OpenSOALab provides the capability to combine forensic logs. Clocks on servers are synchronized with ntpd to millisecond resolution.

Figure 1 contains a graphic of the hotel and currency exchange components of OpenSOALab used in the experiments. OpenSOALab includes two hotel brokers and three currency exchange brokers. It can be deployed over multiple nodes up to a total of 13 if all the banks and hotels are on different machines. It currently utilizes a modified version of the nuSOAP library that provides enhanced entries in the forensic logs. Additionally, the system has a noise generator that allows the
user to request a configurable number of currency exchange requests occurring on a specifiable time interval. The noise generator is used to generate the irrelevant transactions from which the elements of the feature of interest must be identified. The experiments, described in the next sections were performed on a single server and a separate client utilizing the OpenSOALab system. A Test Driver program was also available for the first experiment. The Test Driver program generates a currency exchange transaction of interest with a message added before the start of the feature of interest to help determine when the feature started executing.

A user can use either the currency exchange by itself or the combined system. If the user uses the currency exchange only, the source and target currencies and the amount to exchange are specified, this information is conveyed to a broker, and the broker attempts to complete the transaction. It does so by contacting a list of currency exchange houses to get quotes, choosing the best exchange rate, and finalizing the transaction through a settlement house. The currency exchange system also has an authentication subsystem for banks and brokers. The currency exchange system is time-sensitive as a transaction must occur in the time frame during which the exchange rate quote is valid. The system reports success if the transaction is completed, or failure if it is not completed for any reason.

A user using the entire system of hotel reservations and currency exchange would first log on via the logon screen pictured in Figure 1, to gain access to the hotel reservation client. When the client initiates a hotel reservation transaction, a country of interest is chosen and, based upon the country, a hotel reservation broker is selected. The client interactively provides arrival and departure dates, the number of guests, and the maximum price for a room. Based upon these criteria, a set of rooms and their costs is generated by the broker. If the user selects a room to book, the system computes the amount of currency needed in the target currency and invokes the currency exchange system automatically. The two components (the hotel reservation and currency exchange) were developed separately as student projects, and combined to form the composite application.

Figure 1. The Hotel and currency exchange components of OpenSOALab.
The Feature Sequence Viewer had been developed with input and support from Northrop Grumman corporation to analyze pre-SOA distributed software. It was then adapted to the Apache / php / nuSOAP environment. Input to this version of FSV consists of:

1. A log of time-stamped SOAP messages, obtained from the Apache forensic logs of the servers involved in the composite application.

2. One or more pairs of (start-time, end-time) bracketing a time interval when the feature of interest was executing.

The FSV looks at each distinct message (a tuple of source, destination, and message type) occurring in the whole log, and computes a weight called $p_c$, which is the number of times that this message occurred when the feature was executing, divided by the total number of times it occurred in the whole trace [2]. The value of $p_c$ ranges from 0.0 to 1.0; the higher the value the more likely it is that the message is part of the feature. Figure 2 contains a graphic of the interface to the FSV.

![FSV Interface](image)

**Figure 2. FSV interface displaying a series of highly relevant (orange) and less relevant (blue) messages.**

Using the list of messages and their corresponding values of $p_c$, FSV creates a modifiable graphical representation of the sequence of messages generated as the system executes. Some of those messages were part of the feature, and others were not. The FSV provides the capability to adjust the $p_c$ threshold from 0 to 100% to include relatively more or relatively fewer messages. If the FSV’s $p_c$ value is adjusted to 0%, ALL messages (executing components) are displayed. If the threshold is adjusted to 100%, only the messages with highest probability of being in the feature are shown. At intermediate settings, relatively more or fewer messages are displayed. In Figure 2, the orange lines (having a rating of $p_c \geq 0.9$) represent messages that are very likely part of the feature, whereas the blue lines (for which $0.3 \leq p_c \leq 0.4$) correspond to messages...
that are less likely to be part of the feature. The threshold is adjusted by positioning the slider in the middle-left of the window. The color codes in the legend below the slider indicate \( p_i \) values mapped to the color codings in the numbered lines in the left-hand part of the window. The \( \text{SendingTime} \) messages in the setup textbox indicate the approximate start time and ending time of the feature of interest. These two times are provided as user input to the FSV.

3.2. Descriptions of the Experiments

The goal of the experiments was to recover sequence diagrams in the FSV that will help a Software Engineer understand the various software features that execute in the composite application. Apache servers that were configured to enable forensic logging were used for the experiments. SOAP messages were logged and evaluated to trace execution of the features. Noise was introduced into the experiments through a version of a Web client from the currency exchange system that had been modified to accept input for the number of transactions to run, and the time delay (in seconds) between each transaction. Three experiments of increasing complexity were performed. The first experiment involved the currency exchange only without the noise generator running. The second experiment involved use of the currency exchange with the noise generator running. The third experiment included interactive use of the hotel reservation system and the currency exchange with and without the noise generator running.

3.2.1. Experiment 1. The goal of the first experiment was to understand the sequence of events that took place in a simple existing composite application, the currency exchange. The experiment involved carrying out a currency exchange request, with a specified bank, the "Mikado Bank"; specifying the currencies and amount to be exchanged, authenticating brokers, getting a variety of quotes, selecting the best quote, and making the purchase. The transaction that generated this feature was initiated with a Test Driver page that provided a time stamp for the start of the transaction. Experiment 1 had 2 test runs:

- a single currency exchange transaction with Mikado Bank
- three currency exchanges initiated by human interaction - the first and third were with other banks and the second was with Mikado Bank. This protocol guaranteed that no interleaving of messages would occur.

3.2.2. Experiment 2. The second experiment was similar to the first but with background noise added from a separate client. As previously described, these were transactions that were irrelevant to the feature of interest. The extraneous transactions were created with the configurable noise generator. Experiment 2 required two initial studies, the first to calibrate the noise generator, and the second to determine how closely the client and server machine clocks could be synchronized. Calibration of the noise generator was necessary to determine how many noise transactions were necessary to ensure that noise occurred temporally both before and after the feature of interest, and to determine if noise transactions would interleave with feature of interest transactions. It was determined that both relevant and irrelevant messages interleaved. The Network Time Protocol (ntpd) server at the University of West Florida was used to synchronize the clocks on the client and the server, and a synchronization error of approximately 0.1 msec was achieved. The time synchronization permitted easier estimation of the start time of the feature of interest, since start time for the test driver which produced the feature to be observed was noted from the client’s clock and the forensic log was generated on the server.

3.2.3. Experiment 3. The hotel system was utilized in conjunction with the currency exchange for the final experiment. The system was first set up so that a process of getting prospective hotel rooms, choosing one to book, obtaining a price for the room, and then exchanging the currency executed successfully. An additional feature of the software that permits the user to view all room reservations, was also executed in the first two tests. After running the initial tests, the system was set up so that the attempt was made to change some currency that was not handled by any bank in the system. This scenario simulates a "gap in the system" in the sense that no error existed in the software, which performed as expected. Rather, the problem was that one particular case existed for which no service was available. The idea was to compare normal operation of the combined application against the failure case and assess any differences in the messages that occurred.

Since Experiment 3 involved using both the reservation system, which has two features, and the currency exchange system, this experiment involved the most extensive test. This test involved substantial human interaction (and hence, human time) with the system to:

- Choose a country and city for travel, and specify a customer name
- Specify a maximum room charge, arrival and departure dates, and number of guests
- Select a room from among those that meet the criteria
- Exchange dollars into the currency required for the reservation.
The goal of Experiment 3 was the identification of two features that were named as follows:

1. “book room” - comprised of two sub-features:
   - “make a reservation” - get customer information, identify rooms, and select a room.
   - “exchange currency” - for the price of the room. Necessary to complete feature “book room”
2. “view all reservations” - retrieve and display all existing reservations.

The tests and protocols for Experiment 3 were:

Test 1: Run both features, no noise, successful currency conversion.
Test 2: Run both feature, no noise, no valid currency exchange for "lira."
Test 3: Run “book room” only, with noise, successful currency conversion.
Test 4: Run “book room” only, with noise, no valid currency exchange for "lira."

4. Methods

The procedure used to run the tests and place the results in Feature Sequence Viewer was as follows:

1. Stop the apache server and discard the forensic.log file.
2. Start apache to obtain an empty forensic.log file.
3. (Subject to test protocol) Start the noise generator.
4. Execute the test driver (testDriverMarkedLog.htm), running the test cases and recording the start times for the features.
5. (Experiment 3 only) Log into Hotel, book a room and show all reservations.
6. Run the F2FSV.pl script to reformat the forensic log for viewing in FSV.
7. Create a setup file for the start and end times of the features to be located.
8. Run the Feature Sequence Viewer with the converted forensic log as input.

5. Results

5.1. Experiments 1 and 2.

Experiment 1 concluded as expected by identifying all messages in the feature at a value of $p_c = 0.5$. The output of Experiment 1 is shown in Figure 3. Messages typically occurred in pairs with the first one a request to retrieve the WSDL file and the second the actual remote procedure invocation. In Experiment 2, all relevant components of the feature of interest were identified, but due to the fact that some elements of the feature also occurred in the irrelevant transactions, the value of $p_c$ had to be lowered in order to detect those parts of the transaction. Output from Experiment 2 was used in Figure 2 above to illustrate the FSV interface. All messages that were a part of the feature of interest were included, and no irrelevant messages appeared in the output, with the properly selected value of $p_c$.

Figure 3. FSV output: The message trace for Experiment 1.
5.2. Experiment 3.

Each test of both features in experiment 3 involved a total of 17 human interactions with the system. Consequently, the time duration of the experiment was longer than the other experiments. For the tests in which noise was running, enough transactions had to be specified for the noise generator to ensure that it started running before the transaction of interest was initiated and that it continued to run past the time in which the transaction of interest had completed. Consequently, the tests involving noise contained many more transactions than ones without noise. Table 3 summarizes results of this experiment in terms of features, relevant messages, irrelevant messages and values of $p_c$ that provided the best extraction of relevant messages from irrelevant ones.

<table>
<thead>
<tr>
<th>Test</th>
<th>Feature</th>
<th>Irrelevant Messages</th>
<th>Relevant Messages</th>
<th>Total Messages</th>
<th>Optimal $p_c$ value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>41</td>
<td>54</td>
<td>95</td>
<td>$p_c = 0.68$</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>81</td>
<td>14</td>
<td>95</td>
<td>$p_c = 0.51$</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>40</td>
<td>44</td>
<td>84</td>
<td>$p_c = 0.68$</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>68</td>
<td>16</td>
<td>84</td>
<td>$p_c = 0.51$</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>744</td>
<td>56</td>
<td>800</td>
<td>$p_c = 0.51$</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>771</td>
<td>42</td>
<td>813</td>
<td>$p_c = 0.71$</td>
</tr>
</tbody>
</table>

Table 1. Results for the tests in Experiment 3.

Figure 4 contains a graphic of Test 1, Feature 1. This feature, “book room,” was the one with all the user interactions with the system. The first cluster of messages, appearing in the top-left of the graphic (messages 1-24), were all associated with setting up the customer and identifying the hotel room to be reserved. The second cluster of messages, appearing in the bottom-right of Figure 4 (messages 25-42), were all part of the exchange of currency. The last two messages (43 and 44) were the confirmation messages indicating that the currency had been exchanged and the room was booked. All messages associated with viewing all the reservations were excluded from this view with a $p_c = 0.67$. Figure 4 revealed that available rooms meeting the search criteria were retrieved one at a time, each with a separate, dedicated SOAP request-response. These were messages 1-16 where 1-8 were to the first hotel broker and messages 9-16 were to the second hotel broker. The FSV revealed this design flaw, which had been unknown previously.

Figure 4. Output of Test 1, Feature 1 showing sub-features “make reservation” (top left cluster) and “exchange currency” (bottom right cluster).
Figure 5 contains a graphic of Test 1, Feature 2, which is the “view all reservations” feature. It was produced with the same input message data as Figure 4. The only difference between the two graphics was a different specified start and ending time during which the feature of interest occurred and an adjustment to \( p_c \) to find the optimal setting to maximize signal to noise. Figure 5 reveals several irrelevant messages and was produced with a \( p_c = 0.48 \). Messages 1-24 were actually generated by the Test Driver, and were noise. All messages associated with making the reservation and exchanging currency were excluded from this view with a \( p_c = 0.51 \). By lowering the \( p_c \) value sufficiently, all the messages in the test would eventually have been displayed. Figure 5 revealed a design defect that was similar to the one identified in Figure 4: separate, individual SOAP messages to retrieve each of the current reservations.

Results from the last two tests in Experiment 3, in which the noise generator ran continually, were similar to those pictured in Figures 4 and 5. The main, and very important difference was that, rather than extracting the feature from a limited number of extraneous messages, the features were extracted from in the midst of 800 messages. A graphic of this output is not easily included as it would consume several running pages and still be only minimally readable.

Experiment 3 produced two interesting results. The first was not an error, but it illustrated a potential use of the FSV in finding a significant inefficiency in the system. The sequence graph made clear the fact that the hotel system issues separate SOAP calls that get available rooms and reservations one at a time—a separate call for each reservation or booked room. Those calls can be seen as messages 1-16 in Figure 4 (getting available rooms) and messages 25-40 in Figure 5 (viewing all reservations). The second interesting feature of the reservation system that was revealed was the transaction providing the confirmation of the hotel booking in test 4 after the currency wasn’t exchanged. Data in the system was modified to allow selection of a hotel for which no bank in the system could exchange the currency. When the program ran, all the banks properly reported that they could make the exchange, the hotel system detected that no currency was changed and reported that result. However, the hotel system then proceeded to create the reservation and provide a confirmation number. The students who developed the system did not anticipate that their code would be changed in a way as to preclude successful completion of the exchange, and did not test for success or failure. Clearly, no confirmation number should have been issued.

Figure 5. Output of Test 1, Feature 2 showing the feature “view all reservations.”
6. Acknowledgments:

Work described in this paper was partially supported by the University of West Florida Foundation under the Nystul Eminent Scholar Endowment. Development of the Feature Sequence Viewer was supported by Northrop Grumman Corporation via the Software Engineering Research Center (http://www.serc.net).

7. Summary and Conclusions

This report contains a description of several tools and experiments that were performed to demonstrate a dynamic approach to the location of features in composite applications. After a review of literature pertaining to the topic, OpenSOALab, an Apache/php-based composite application testbed, and FeatureSequenceViewer, a tool providing a modifiable graphic display of the sequence in which features in a system execute, were described. These along with a noise generation capability and test driver were used to perform several experiments in feature location based upon SOAP messages. Results, which were displayed and manipulated in the Feature Sequence Viewer, included demonstration of the capability to extract a simple feature from minimal background noise as well as the capability to identify a feature of interest involving 42 messages from among more than 800 total messages. The FSV facilitated identification of an error in the hotel reservation system that was not previously known. The FSV also revealed inefficiencies in the system that would have been difficult to detect through other means. The ability to detect inefficiencies of the type identified by this system would be very useful in identifying QoS bottlenecks in composite applications.

8. References


