Toward Web Service Dependency Discovery for SOA Management

Sujoy Basu
HP Laboratories
1501 Page Mill Road
Palo Alto, CA 94304, USA
001 650 236 2044
sujoy.basu@hp.com

Fabio Casati
University of Trento
Via Sommarive 14
38050 Povo (TN), Italy
0039 0461 88 2044
casati@dit.unitn.it

Florian Daniel
University of Trento
Via Sommarive 14
38050 Povo (TN), Italy
0039 02 2399 3667
daniel@dit.unitn.it

ABSTRACT

The Service-Oriented Architecture (SOA) has become today's reference architecture for modern distributed systems. As SOA concepts and technologies become more and more widespread and the number of services in operation within enterprises increases, the problem of managing these services becomes manifest. One of the most pressing needs we hear from customers is the ability to "discover", within a maze of services each offering functionality to (and in turn using functionality offered by) other services, which are the actual dependencies between such services. Understanding dependencies is essential to performing two functions: impact analysis (understanding which other services are affected when a service becomes unavailable) and service-level root-cause analysis (which is the opposite problem: understanding the causes of a service failure by looking at the other services it relies on). Discovering dependencies is essential as the hope that the enterprise maintains documentation that describe these dependencies (on top of a complex maze and evolving implementations) is vane. Hence, we have to look for dependencies by observing and analyzing the interactions among services.

In this paper we identify the importance of the problem of discovering dynamic dependencies among Web services and we propose a solution for the automatic identification of traces of dependent messages, based on the correlation of messages exchanged among services. We also discuss our lessons learned and results from applying the techniques to data related to HP processes and services.

Keywords

SOA Management, Service Dependency, Discovery.

1. INTRODUCTION

Discovery of dependencies among components of large distributed systems is an important technology for management software. In enterprise IT systems, many tools for discovery exist that find the relationships among components, e.g., by looking into configuration files. These tools can discover for example that an application running in a J2EE application server depends on an Oracle server running on another host. It proves very effective when discovering *static* dependencies among coarse-grained systems.

However, customers are increasingly asking for the capability to discover *dynamic* dependencies among services. In fact, many business applications today are based on a service-oriented architecture (SOA), which implies that they are composed of loosely-coupled, reusable services. When a service has a failure or performance degradation, all other services that depend directly or indirectly on this service might be impacted. It is important to understand what these dependencies are, so that management tools can display and alert users about the business impact of failures and performance degradations. Furthermore, knowledge of dependencies considerably simplifies service-level root-cause analysis, that is, trying to understand the origin of a failure.

The dependencies can be explored in tools such as HP OpenView SOA Manager, and the performance metrics of all the dependent services are captured. However, currently the dependencies must be explicitly specified in SOA Manager. This works in theory when enterprises enforce change management processes strictly. Any change in dependencies of a SOA business application on the underlying services can then be captured in a database and updated in SOA Manager. In practice, these processes are not always followed. Furthermore, the specification of the dependencies may include some inaccuracies. This is why customers have repeatedly requested to endow SOA Manager with a dependency discovery module, which derives dependencies by looking at message exchanges among services.

In this paper we present a module that automatically analyzes service execution data to discover dynamic dependencies among services. The problem is far from trivial as it requires understanding correlations among message exchanges between services. For example, if we observe that service A invokes service B, and service B invokes service C, in which cases can we say that the second invocation is caused by the first and that there is, therefore, a dependency between A and C? This is already a problem in this simple scenario with three services, and it becomes a much more complex problem when there are multiple services used in different combinations and delivering different kinds of functionality. Hence, the problem is important from a business standpoint and is challenging from a research perspective.

This paper is structured as follows. In Section 2 we describe our reference architecture and formalize the problem addressed in this paper. In Section 3 we position the problem with respect to related work, while in Section 4 we describe our own approach to the discovery of dependencies among Web services, and we discuss the underlying algorithm in more detail. In Section 5 we report on our first experiments and the results obtained so far. Finally, in Section 6 we conclude the work and provide an outlook over ongoing and future work.

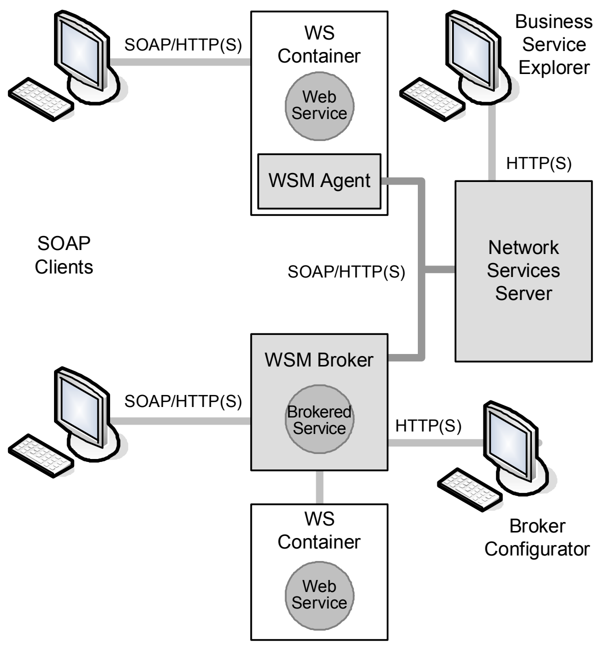

Figure 1 SOAM deployment architecture for resource management.

2. CONTEXT OF INVESTIGATION AND PROBLEM STATEMENT

The discovery of dependencies between Web service executions in distributed environments is a challenging problem in general. Due to the wealth of possible different software and hardware infrastructures and the considerable number of different protocol specifications that can be used in a specific implementation, the dependency discovery problem comes in a variety of different flavors. In order to be able do define the problem under study, we thus first describe the platform we aim at extending, i.e. HP's OpenView SOA Manager [6].

2.1 Reference Architecture

Figure 1 describes the reference architecture of our dependency discovery platform, where our initial concern is the capturing of *message traces* and the identification of communicating *components* (i.e. the services).

Message traces can easily be captured by the logging mechanism already provided by SOA Manager. SOA Manager intercepts any message sent to a service it manages. This can be done in two different fashions: (i) by means of a handler or *agent* running in supported Web service containers, such as BEA WebLogic and .Net containers, or (ii) by means of suitable interception proxies (*brokers*) for services running in all other environments. An agent runs code provided with the SOA Manager product and has access to message headers. Many fields from the headers are logged in an audit database. Brokers on the other hand are executed outside the container and must intercept all messages sent to the service they manage, so that the message headers can be similarly logged. Clients and other services invoking the brokered service must be configured to use the brokered endpoint rather than the original service endpoint.

Intercepting messages and their responses using either of these two approaches is primarily for the purpose of computing metrics such as response time or request frequencies. However, the intercepted message traces are also stored in a centralized database to enable auditing. Dependencies among services can thus be discovered by analyzing these message traces.

As our study aims to identify dependencies based on real log data and without requiring SOA Manager to provide any additional logging feature, there are a few considerations that need to be taken into account and that constrain the possible solution space:

- We cannot rely on any protocol-specific *correlation* or *addressing* information possibly encoded in the body of intercepted messages, as for example provided by WS-Coordination or WS-Addressing. Although such protocol extensions are good practice, they still lack widespread use. The result is that, given any set of messages, we cannot determine in general if two messages belong to the same conversation. Even if WS-Coordination or WS-Addressing are used, the scope of the coordination typically encompasses only the interaction among a few services, and does not follow all dependencies (indeed, it would otherwise violate the very same loose coupling principle on which SOA are based).
- We cannot analyze the *body* of traced messages, because although the logging mechanism does support the logging of message bodies, this feature is in general disabled by system administrators: the load is often too much unless in trivial lab tests.
- For a given message, in general we only know the destination service/operation and the network address of the node that hosts the source service, unless WS-Addressing is used. If WS–Addressing had widespread support, this would not be a problem. But since this is not yet the case (and it is not clear when this standard will become widely used), the availability of only the IP address of the sending service complicates the problem of inferring dependencies between messages, since multiple services can be located at an individual IP address.
- Finally, we observe that logged data only contains messages sent by or directed to services *managed* by SOA Manager. Possible service dependencies outside the managed environment cannot be derived.

2.2 Problem Statement

Given the above considerations and an assignment of Web services $S_i \in S$ ($1 \leq i \leq n$, S being the set of Web services managed by SOA Manager) to the nodes $N_j \in N$ ($1 \leq i \leq m$, N being the set of computing nodes managed by SOA Manager) by means of the a function $loc: S \rightarrow N$, we can formalize a message M as tuple $M=(N_s,S_d,t)$, where N_s is the IP address of the source node, S_d is the destination service, and t is the time the message is received by S_d .

Being $L=(M_1,M_2,\ldots,M_{last})$ the chronologically ordered message log with $t_1<t_2<\ldots<t_{last}$ and being t_{last} the time corresponding to the last logged message, our problem of discovering dependencies among Web services can be reformulated as the discovery from the log data of all those message traces $T=(M_1,M_2,\ldots,M_k)$ with $t_1<t_2<\ldots<t_k$, $k \leq t_{last}$ and $S_{d,i}$ located on $N_{s,i+1}$ and M_{i+1} generated by $S_{d,i}$ in response to M_i for all $1 \leq i \leq k-1$. In short, we need to identify traces of messages that are causally (e.g. functionally) dependent.

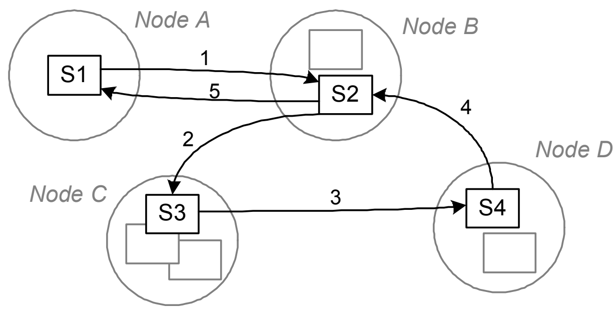

Figure 2 Example sequence of message exchanges among managed services. Circles represent computing nodes, rectangles represent Web services.

Consider for example the messages exchanged among the Web services shown in Figure 2; messages from a service S_1 to a service S_2 are represented by arrows, and labels correspond to the temporal order of the messages. There are four nodes that host different numbers of Web services and, in particular, four services that exchange messages in a chronologically ordered and causal logic that represents the message trace $T=(M_1,M_2,\ldots,M_5)$ to be discovered. Typically, the generation of such a message trace is driven by a particular coordination agreement (i.e. a protocol) among the cooperating services, which we however assume unknown.

More precisely, let us assume that in Figure 2, message 2 from service S_2 to S_3 is the result of message 1 from S_1 to S_2. However S_2 might have spent time on computation before sending message 2 to S_3. It is also likely that S_2 is multithreaded and therefore has received and sent other messages between receiving message 1 and sending message 2. Hence we have to solve the problem of identifying the correlated messages 1 and 2 (and hence a dependency of S_1 on S_3) from a log where they are not adjacent entries. In the absence of support for WS-Addressing, when message 2 is intercepted at S_3, only the IP address of node B can be logged. The fact that S_2 sent the message is not known. There is no general solution for intercepting the message when it leaves node B since S_2 may not be running in a J2EE or .Net container (e.g. because it is implemented in C++).

3. RELATED WORK

Research works. The illustrated problem is not a trivial one. There are several works in literature that address similar issues, prevalently in the area of *sequence* or *pattern mining*.

Itemset mining, as an instance of sequence mining technique, is used above all in marketing and CRM applications to identify repeated patterns in a sequence of (business) transactions. In [5] the authors describe for example an interesting approach to sequential pattern mining (GSP) that also leverages user-defined taxonomies during the mining process and outperforms their previously proposed AprioriAll algorithm. For an overview of frequent itemset and association rule mining the reader is referred to [7].

Unfortunately, itemset mining techniques are not applicable in our case, as we do not have any notion of confined and identifiable transactions. We are given a flat, sequential log stemming from a distributed computing environment where possible correlated communications, i.e. conversations, are not tagged by a unique conversation identifier, and their message sequences are typically interleaved in the log file.

Our problem thus resembles more closely the one addressed by another sequence mining technique, i.e. *string* or *episode mining*. String mining (see for example [8]) is heavily adopted in bioinformatics, while episode mining (see for example [4] and [9]) rather concentrates on large event sequences, e.g. in the telecommunications domain.

Especially the work presented in [4] could be promising in our context, but there are two main constraints that differentiate our problem from the one studied by the authors in [4]: (i) a log entry (i.e. a message sent from service S_1 to service S_2) does not uniquely identify the source service, as we only are given the source IP address, at which there might be located several different services possibly generating the message; (ii) we are in presence of both short-running conversations and long-running business processes, which makes it difficult to identify suitable time windows for the mining process and, thus, heavily would increase computation times.

In [2] the authors concentrate on performance debugging in distributed systems, a conceptually similar problem to the one discussed in this paper. They propose two interesting approaches, which are not based on mining techniques: an algorithm based on the nesting of request and response messages in RPC style communications, and an algorithm based on signal processing for free-form message-based communications. While the former algorithm is not applicable to our domain, the latter, again, presents the problem of sizing a suitable time window a priori in presence of long-running business processes. In [3] the authors present their *message-linking algorithm*, an evolution of the work presented in [2], which assumes that causal delays between an incoming and an outgoing message follow an exponential distribution. If the time difference between an incoming and an outgoing message exceeds four times the average delay that can be derived for the two messages from the event log, no causal dependency is assumed anymore; this corresponds to adopting a different time window for each pair of incoming and outgoing messages at each node of the system.

Competitive approaches and products. The main constraint imposed by our reference scenario is that we cannot run agent code in all service containers, which would allow us to tag correlated messages with a unique identifier, and dependent messages could be identified with certainty on the basis of the identifier. This is however the approach taken by several vendors in the SOA management space, such as Actional (Sonic Software) and IBM. The patented solution by Actional [10], for example, is based on agents that operate on the application protocol level and have visibility of both inbound and outbound messages. Agents tag messages with correlation data (both in input and in output) and feed a proper Agent Analyzer module with the enriched message records, which is then able to accurately trace all dependencies that exist among the running services.

This approach however does not work for Web services that do not run in containers, as might be the case for legacy applications to which Web service interfaces have been added. Instead, it is our goal to also support Web services that run outside service containers.

4. DEPENDENCY DISCOVERY

We now present our approach to dependency discovery. The problem of dependency discovery is complex because in many situations there is a large number of invocations on a fairly large number of services. Hence, if we restrict our approach to the simplistic determination of checking that when service A is invoked, then service B is invoked shortly afterwards, we would be out of luck, since both A and B are frequently invoked and it is not possible to say that two given invocations are dependant. The more frequent service invocations are, the more complex the problem becomes, and in general it is impossible to be "certain" about a dependency. Hence the philosophy we have taken in this work is to find a set of "suspicions" (rather than evidences) that two services are dependant. When we have sufficient suspicions we conclude that a dependency exists.

Specifically, our approach to the discovery of dependencies among Web services according to the definitions given in Section 2.2 and in consideration of the works discussed above is composed of four consecutive steps:

1. Inference of a *causal dependency* within message pairs in the log, where the first message is received at the service node from which the second message originates. In this part we adopt and combine different techniques to detect potential dependencies.
2. Construction of a *probabilistic dependency graph* as concatenation of all identified dependencies between pairs of messages by taking into account the assignment of services to nodes $loc{:}S \rightarrow N$. Edges are labeled with a confidence level, which is the probability of the identified dependency.
3. *Pruning* of the dependency graph by applying a user-specified threshold T_p to the probabilities associated with the edges of the graph, thus simplifying the graph and keeping only "relevant" edges.
4. Construction of *paths* from the pruned graph and mining of the audit log to decide which of the paths indeed occur with a frequency greater than a given threshold T_f.

Further details on these four phases are given below.

4.1 Inference of Causal Dependencies

Inferring causal dependencies within message pairs is the first and basic step toward the identification of entire traces (i.e. paths) of dependent messages. In our current work we investigate three different dependency identification algorithms that leverage the following ideas in order to associate a *dependency probability* to pairs of messages:

- Occurrence frequency of logged message pairs;
- Distribution of service execution times;
- Histogram of execution time differences.

In this paper we show results based on all three approaches. However for the second and third approach, we have not yet automated the selection of dependencies since thresholds need to be tuned. We are currently performing this tuning using SOA applications from different domains, and will present the results in the camera-ready version of the paper.

4.1.1 Occurrence Frequency

The first approach is based on the frequency of the occurrence of message pairs in the log data, i.e. it is based on the *conditional probability* that a message M_2 can be found in the log data, knowing that M_1 has been found.

We fix a time window size w $(1 \leq w \leq t_{last})$, which corresponds to the limit on the execution time of a service from the time a message is received by the service until a dependent message is sent out by the service. If the conditional probability of message M_2 appearing within the time window whenever message M_1 occurs in the database exceeds some threshold, we infer that message M_2 is dependent on message M_1.

The detailed algorithm is described by the following pseudo-code:

Algorithm:
Get conditional message dependency probabilities

Initialize $i=1$.

Initialize an empty set CM of message counters.

Scan the audit log data L in order of increasing timestamps, and execute for each message $M_i \in L$, $1 \leq t_i \leq t_{last}$:

> Step 1: Initialize the time window W_i corresponding to message M_i with $W_i=(M_{i-w},\ldots,M_i)$, $W_i \subseteq L$ and $M_j \in L$ for $t_{i-w} \leq t_j \leq t_i$.
>
> Step 2: Initialize an empty set S_i of message signatures and an empty set CP_i of message counters. A message signature of a message M consists of a hash function computed over sender, receiver and invoked operation corresponding to M.
>
> Step 3: Start at the earliest message in the time window if M_i is not present in the time window ($t_p = t_{i-w}$). Alternately start at the message following the most recent occurrence of M_i in the time window (t_p). Execute for each message $P_{i,j}$ ($p \leq j < i$) from the starting point to the end of the time window:
>
> > Step 3a: If there is a potential causal dependency from $P_{i,j}$ to M_i indicated by the destination service of $P_{i,j}$ being located at the same node at which M_i is generated, compute the signature $S_{i,j}$ of $P_{i,j}$.
> >
> > Step 3b: If $S_{i,j} \in S_i$, set $i=i+1$ and $CP_{i,j}= CP_{i,j}+1$ and go to Step 3.
> >
> > Step 3c: Add signature $S_{i,j}$ to S_i, set $CP_{i,j}=1$ and add the counter $CP_{i,j}$ to the set CP_i. Got to Step 3.
>
> Step 4: If $CM_i \in CM$ set $CM_i=CM_i+1$, else set $CM_i=1$ and add the counter CM_i to the set CM.
>
> Step 5: Increment $i=i+1$ and go to Step 1.

After the scan of the log database is completed, the conditional probability $P(P_{i,j}M_i/P_{i,j})$ is easily computed as $P(P_{i,j}M_i/P_{i,j})=CP_{i,j}/CM_j$

4.1.2 Execution Time Distribution

The second approach is based on the *statistical distribution* of service execution times from the instant a message is received until a dependent message is sent out.

We assume that service execution times follow a specific statistical distribution (e.g. a normal distribution or an exponential distribution, as suggested in [3]). To verify whether a specific message pair is indicative of dependency, a distribution test will be applied to the time differences between the specific incoming and outgoing messages, which can be derived from the message log data. We look in particular for normal distributions. Only those message pairs whose distribution of time differences fits the statistical distribution with a confidence level exceeding a predefined threshold will be considered dependent. Several statistical approaches and tools to find distributions are available so our goal is not to create a new approach to discovering distributions here.

Multiple instances of a message pair may be interleaved; hence, identifying an instance of a message pair will involve heuristics like skipping a certain number of occurrences of messages or deriving a suitable maximum time window to consider. The preliminary tests performed on some real service log data (see Section 5) confirmed our initial intuition that in *absence* of interleaved message pairs, dependent messages yield a normal distribution, while in *presence* of interleaved message pairs, dependent messages yield an exponential distribution.

4.1.3 Time Difference Histogram

The third approach is based on the computation of a *histogram* of the time differences for all instances of the message pairs, without assuming any predefined statistical distribution a priori.

The presence of a small number of consecutive buckets of the histogram with counts much higher than the average count across all buckets is a likely indicator of the messages having a dependency. As we will show in the following section, this technique is especially suited to the human user inspecting the service log data and looking for dependencies.

The current weakness of the previous two approaches is that we have not yet performed a thorough data analysis on different datasets to be able to state with precision which thresholds are appropriate. Such a thorough analysis is underway and will need to be completed before these two additional techniques for dependencies are included in the tool. However, data confirms the intuitions described above in terms of distributions and histograms.

4.2 Creating the Dependency Graph

Once causal dependencies within pairs of messages have been identified, a *probabilistic dependency graph* is created with nodes corresponding to services and edges corresponding to messages. The construction of the dependency graph is based on inferred dependencies among messages, the association of messages to services, and the assignment of services to nodes $loc{:}S \rightarrow N$.

Since all the techniques illustrated in the previous step produce results of probabilistic nature, each edge or message of the graph is labeled with a probability that expresses the confidence level of the inferred dependency and the probability of the message to source service assignment performed during the construction of the dependency graph.

4.3 Pruning the Dependency Graph

The so created probabilistic dependency graph summarizes the associations of probabilities to messages, of messages to services, and of services to nodes that could be derived from the log data. In order to discriminate paths in the dependency graph with low probabilities, we now *prune* the edges of the graph by applying a predefined threshold T_p. Varying the threshold determines how selective we are about the dependencies that are found. A low threshold only generates dependencies of which the system is more certain. A high threshold implies more dependencies, but of which we can be less confident. Graphically, this translates into a slider that shows dependencies at the changing of the threshold in the slider.

4.4 Identifying Frequent Paths

From the pruned dependency graph created in the previous step, we now identify all possible *paths* representing traces of dependent message exchanges among the managed services, i.e. web service conversations. The identification of a conversation requires computing the effective support of the paths by inspecting once again the audit log data. The audit database is thus mined to decide which of the listed paths occur with a frequency exceeding a predefined threshold T_f; paths with a high enough support represent likely conversations, paths with a low support are discarded.

At the end of these steps, we obtain the graph of dependencies. Since composite applications built on SOA principles use only some of the services as entry points, we can assume that these services are provided as input to the dependency discovery process. In the previous step, only the frequent paths rooted in these services need to be identified. These frequent paths together form trees rooted in these entry points. These trees are provided as dependencies to SOA Manager. Any alert in a monitored service is then propagated along the edges of the trees, and helps in quickly identifying the service where the root-cause of a problem lies.

5. EXPERIMENTS AND RESULTS

Our first experiments with dependency discovery focus on the derivation of the conditional probabilities as described in Section 4.1.1. We based our analysis on HP-internal data about the execution of business processes invoking various services within HP. In the following we will first describe how we converted this data into a format that is compliant with our algorithm (i.e. compatible with SOA Manager audit log data), then we will present the results of our experiments.

5.1 Data Collection and Preparation

Our data set consists of 2 tables of process execution data, which through some elaborations can be transformed into Web service audit log data. The first table lists all the nodes (activities) in the workflow graphs. Each entry has a node ID and a descriptive name. Since there are multiple workflows, each entry also has a flow ID representing the workflow to which the node belongs. The second table lists all the instances of the workflows described in the first table, that were executed between 6^{th} and 15^{th} July, 2005. Each entry in this table has the node ID from the first table, a begin and an end timestamp of the activity, a unique node instance ID, and a flow instance ID that correlates all node instances of the same workflow.

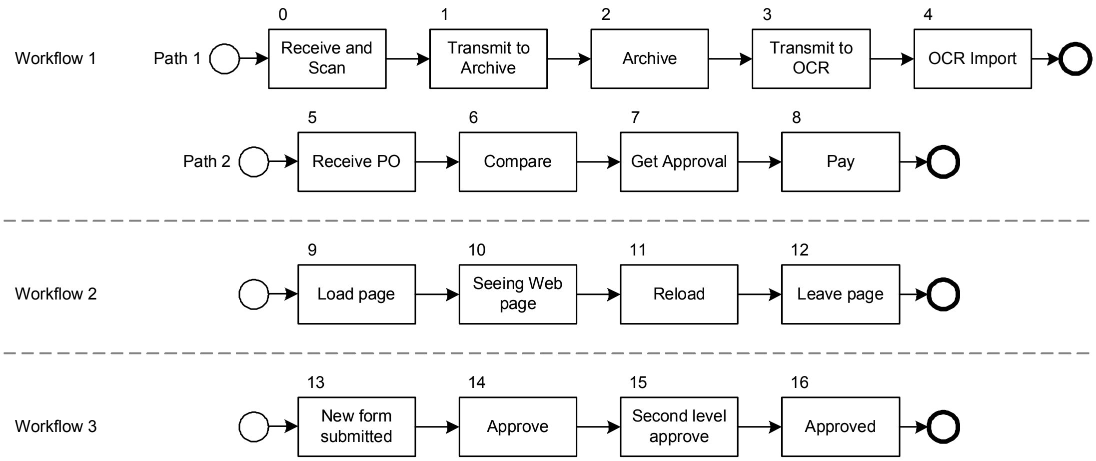

Figure 3 Process models underlying the workflow data used for the experiment. Workflow activities are numbered to characterize messages by means of their source and destination activities.

To preprocess this data set and generate the message trace that our algorithm can accept as input, we analyzed the first table to identify all workflows. Figure 3 shows the result of this analysis in form simplified BPMN[1] process specifications. The log data contains data coming from three different workflows, where workflow 1 is further characterized by the presence of two different paths.

We observed by extracting all entries for a flow instance ID (i.e. a workflow) in the second table that the end timestamp of a node was the same as the begin timestamp of the successor node, according to the activity order depicted in Figure 3. This was verified for several flow instance IDs and could thus be expected to be generally true for this dataset since it consists of workflows. We used this peculiarity as the basis for generating message traces.

Each entry in this message database consists of a pair of nodes and the timestamp at which the first completed and the second started. Since the SOA Manager audit database would contain the actual message sent from the first to the second service, we create that field of the entry by taking a hash of the two node names. In the following we characterize a message from a service S_1 to a service S_2 as a tuple (1,2) of their identifiers (cf. Figure 3 for the identifiers of the activity instances in our experimental data.

5.2 Experimental Results

Using the occurrence frequency approach of section 4.1.1, we were able to identify all the dependencies existing among the messages in the log data prepared in the previous step, i.e. we were able to identify all the dependencies existing among the workflow activities depicted in Figure 3. Table 1 shows the probabilities that could be computed and that represent the confidence level with which an outgoing message is causally dependent on an incoming message.

The dependencies reported in Table 1 match what occurs in the actual implementation (see Figure 3). The technique based on conditional probabilities is therefore able to find dependencies that go beyond the obvious and direct ones.

Incoming message	Outgoing message	Probability
(6,7)	(7,8)	0.83
(5,6)	(6,7)	0.89
(2,3)	(3,4)	0.94
(0,1)	(1,2)	0.95
(1,2)	(2,3)	0.95
(9,10)	(10,11)	1.00
(10,11)	(11,12)	1.00
(13,14)	(14,15)	1.00
(14,15)	(15,16)	1.00

Table 1 Identified dependencies with confidence levels.

We have mentioned earlier that the techniques in section 4.1.2 and 4.1.3 currently depend on the user's intuition for the thresholds to be set correctly. We provide two examples in Figure 4 and Figure 5. Figure 4 shows the application of the execution time distribution test from Section 4.1.2 to our test data. As can be seen in the figure, the values of the test for dependent messages are clearly lower than the values of the test for independent messages. This proves that the distribution test is suited for the automated identification of dependencies, yet appropriate thresholds need to be defined. Figure 5 shows the application of the histogram technique from section 4.1.3. We have highlighted dependent messages, namely the transmission of a document to the OCR system as a result of the transmission of its scanned image to the archival system. The exponential shape of this histogram is characteristic of dependant messages with interleaved message pairs. For independent messages, the histogram typically depicts bars of fairly uniform size. We need to train a classifier to determine whether the messages corresponding to a histogram should be classified as dependant or independent.

[1] Business Process Modeling Notation

Nodes	Exponential test result
Rejected >> Approve >> Rejected	18122
Rejected >> Approve >> Second level approve	53023
Rejected >> Approve >> Approved	51936
Rejected >> Approve >> New form submitted	69767
Rejected >> New form submitted >> Rejected	18554
Rejected >> New form submitted >> Approve	71728
Rejected >> New form submitted >> Second level approve	52092
Rejected >> New form submitted >> Approved	52648
Approve >> Rejected >> Approve	18246
Approve >> Rejected >> New form submitted	18530
Approve >> Second level approve >> Approve	162019
Approve >> Second level approve >> Approved	149014
Approve >> Second level approve >> New form submitted	161661
Approve >> Approved >> Approve	160847
Approve >> Approved >> Second level approve	139799
Approve >> Approved >> New form submitted	160478
Approve >> New form submitted >> Rejected	67874
Approve >> New form submitted >> Approve	283317
Approve >> New form submitted >> Second level approve	210528

Nodes	Exponential test result
Receive PO >> Compare >> Get Approval	3710
Transmit to Archive >> Archive >> Transmit to OCR	3705
Compare >> Get Approval >> Pay	3729
Receive & Scan >> Transmit to Archive >> Archive	3677
Archive >> Transmit to OCR >> OCR Import	3662
Receive PO >> Compare >> Get Approval	1385
Transmit to Archive >> Archive >> Transmit to OCR	1345
Compare >> Get Approval >> Pay	1387
Receive & Scan >> Transmit to Archive >> Archive	1276
Archive >> Transmit to OCR >> OCR Import	1338

Figure 5 Two screen shots of the execution time distribution test (see Section 4.1.2). The first image shows the values of the test performed on independent messages, the second image shows the results of the test performed on dependent messages. The lower the value of the test, the better the measured time differences follow the chosen reference distribution.

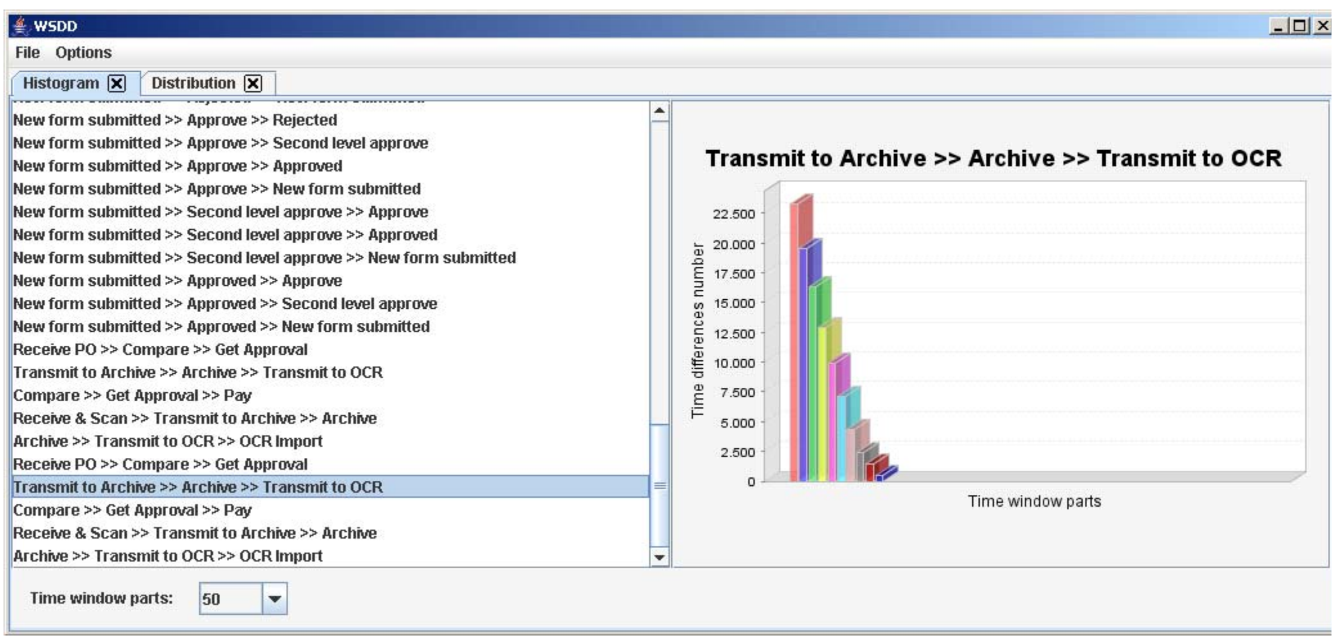

Figure 5 In this time difference histogram (see Section 4.1.3), we have highlighted two dependent messages. In the histogram at the at the right hand side one can easily identify the exponential distribution.

6. CONCLUSION AND FUTURE WORK

In our research we focus on the (semi)automated discovery of complete Web service coordination protocols from unstructured message logs, a relevant research topic in SOA-based distributed systems. In [1], we describe how we have implemented a tool that discovers the model underlying the interaction between services, in terms of state machines for each individual service, given a set of correlated message traces. The models for the different services can be composed to derive the model describing the overall interaction, i.e. the coordination protocol. The solution proposed in this paper complements and extends the work presented in [1] by deriving the necessary correlated message traces from unstructured service execution log data.

We have shown that understanding and discovering dependencies in SOA-based distributed systems is however a complex task and that it is not easy (if not unfeasible in certain situations) to derive dependencies with certainty. The quality of the probabilistic dependency models that can be derived from audit logs is strictly related to the quality of the logged data. As a consequence, in the absence of uniquely identifiable message exchanges – as discussed in this paper – sometimes we may only be able to infer dependency with high probability rather than absolute certainty. Despite this uncertainty, in this paper we have shown that it can anyway be possible to derive useful correlation data from such kind of audit log data.

Although in this paper we discussed the problem of dependency discovery among Web services in the context of HP SOA Manager, we would like to emphasize that the problem is general in nature. Unfortunately, situations where only very poor message log data is available are the majority, e.g. due to the use of different vendor technologies or incompatible audit logging policies.

Our future work will be experiments for determining thresholds for the distribution and histogram techniques. We also intend to algorithmically combine these two techniques with the occurrence frequency technique to make the dependency discovery more accurate, and free the user from the task of deciding which technique will work best in a new situation.

7. ACKNOWLEDGMENTS

We would like to thank the SOA Manager product R&D team for many useful discussions.

8. REFERENCES

[1] F. Casati, B. Benatallah, H. Motahari, R. Saint Paul. Protocol Discovery for Web services. In Openview University Association Workshop (OVUA), Nice, France, May 2006.

[2] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, A. Muthitacharoen. Performance Debugging for Distributed Systems of Black Boxes. SOSP'03, Bolton Landing, NY, USA, October 2003.

[3] P. Reynolds, J. L. Wiener, J. C. Mogul, M. K. Aguilera, A. Vahdat. WAP5: Black-Box Performance Debugging for Wide-Area Systems. Proceedings of WWW'06, Edinburgh, Scotland, May 2006.

[4] H. Mannila, D. Rusakov. Decomposition of Event Sequences into Independent Components. First SIAM Data Mining Conference, April 2001, Chicago, IL, USA.

[5] R. Srikant, R. Agrawal. Mining Sequential Patterns. Generalizations and Performance Improvements. Proceedings of the 5th International Conference on Extending Data-base Technology (EDBT), 1996.

[6] Hewlett-Packard Company. SOA Manager. http://h20229.www2.hp.com/products/soa/

[7] B. Goethals. Survey on Frequent Pattern Mining. University of Antwerp, Belgium, 2006. http://www.adrem.ua.ac.be/~goethals/software/

[8] S. D. Lee, L. De Rade. An Efficient Algorithm for Mining String Databases Under Constraints. KDID 2004, pages 108-129.ù

[9] N. Méger, C. Rigotti. Constraint-based mining of episode rules and optimal window sizes. In Proceedings of the 8th European Conference on Principles and Practice of Knowledge Discovery in Databases, Pisa, Italy, 2004, pages 313 – 324.

[10] Actional/Progress Software Corporation.. Business Process Visibility. White paper, 2006. http://www.actional.com/