An Extensible Architecture for Run-time Monitoring of Conversational Web Services

Konstantinos Bratanis
South East European Research Centre
Research Centre of the Univ. of Sheffield and CITY College
Thessaloniki, Greece
kobratanis@seerc.org

Dimitris Dranidis
Department of Computer Science
CITY College, International Faculty of the Univ. of Sheffield
Thessaloniki, Greece
dranidis@city.academic.gr

Anthony J.H. Simons
Department of Computer Science
University of Sheffield
Regent Court
211 Portobello Street
Sheffield S1 4DP, UK
a.simons@dcs.shef.ac.uk

ABSTRACT

Trust in Web services will be greatly enhanced if these are subject to run-time verification, even if they were previously tested, since their context of execution is subject to continuous change; and services may also be upgraded without notifying their consumers in advance. Conversational Web services introduce added complexity when it comes to run-time verification, since they follow a conversation protocol and they have a state bound to the session of each consumer accessing them. Furthermore, conversational Web services have different policies on how they maintain their state. Access to states can be private or shared; and states may be transient or persistent. These differences must be taken into account when building a scalable architecture for run-time verification through monitoring. This paper, building on a previously proposed theoretical framework for run-time verification of conversational Web services, presents the design, implementation and validation of a novel run-time monitoring architecture for conversational services, which aims to provide a holistic monitoring framework enabling the integration of different verification tools. The architecture is validated by running a sequence of test scenarios, based on a realistic example. The experimental results revealed that the monitoring activities have a tolerable overhead on the operation of a Web service.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures—*Service-oriented architecture (SOA)*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Reliability*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Monitors*

General Terms

Theory, Design, Experimentation

Keywords

Run-time monitoring, conversational Web services, monitoring architecture, validation

1. INTRODUCTION

Web services typically reveal their behaviour only as a collection of interfaces. This means that state-related behavioural errors in Web service implementations are hard to detect, relying on the skill of the service provider to validate all the necessary protocols. When this is coupled with the likelihood that the provider will frequently wish to update the service and that the context in which the service executes will also be subject to rapid change, these factors together become a major source for errors, when a service is being invoked as part of a service-based application (SBA). The combination of rapid change and the lack of a behavioural specification make it extremely likely that a Web service will deviate from its expected behaviour at run-time.

Furthermore, the introduction of a defective Web service within an SBA can have a devastating effect that puts the sustainability of the SBA at risk. Hence, being able to verify that the behaviour of Web services conforms to their advertised specifications at run-time is considered critical.

A promising solution to the aforementioned issue is to continuously monitor a Web service for deviations during run-time. Run-time monitoring could involve a formal verification technique for ensuring conformance of the monitored Web service. The monitors operating in parallel with the monitored Web service, observe and record the messages exchanged between the service and a service consumer in order to detect inconsistent behaviour. The intended behaviour of the service needs to be included in the service specification and expressed in a machine-readable form.

In [8], where we have presented a framework for run-time verification for Web services, we have classified stateful services into the non-conversational and conversational; the former accept all operations at all states, whereas the later accept only a specific protocol. We would like to underline that a conversational service may be implemented as a stateless interface with stateful behaviour. For instance, a Web service that exposes an API to access a travel reservation system. Although, the Web service itself is stateless, the reservation system maintains a state, and therefore the service may accept only a specific sequence of invocations. We further distinguished Web services w.r.t. their state modifiability to private-state for services that their state is fully determined by the sequence of previous invocations, and in shared-state for which the state cannot be determined by the previous service invocations. Lastly, we classified services into transient-state for those services that their state is destroyed after the completion of a session, and to persistent-state for the ones that their state outlast the duration of the session.

The focus of the paper is not on a particular monitoring approach, but on the architecture that could be used for integrating different monitoring approaches. Thus, in this paper we present an implementation of a monitoring architecture for conversational Web services. The architecture is built in a modular way and is vendor-agnostic, so that different vendor instantiations can be supported. We distinguish services in several categories, depending on how the session information is stored and handled. We validate the monitoring architecture by performing monitoring on a private and transient state service. Additionally we measure the overhead of monitoring and test the architecture under stress conditions.

The paper is structured as follows. Section 2 discusses about the implementation of the monitoring architecture. Section 3 provides an overview of the modelling formalism. Section 4 presents the evaluation of the monitoring architecture based on a realistic conversational Web service used as an example. Some key findings are discussed in section 5 and this work is contrasted with related work in section 6. In the conclusions we summarise the main points of our work and provide an outlook for future research.

2. IMPLEMENTATION OF THE MONITORING ARCHITECTURE

2.1 Extensible and Open Architecture

The necessity for an open monitoring framework which allows joint monitoring approaches has been also underlined in [5]. The authors support that there exists a wide range of methodologies for monitoring Web services and service orchestrations which are different in the aspects being monitored and the model being utilised. However, most of them manage to solve small fragments of different monitoring aspects and it is therefore necessary to combine the existing approaches, in order to get the positive aspects of different solutions and construct more complete monitoring solutions.

One of the primary objectives of the implemented monitoring architecture was to provide a platform for integrating different monitoring approaches for Web services. Although in this paper we use the architecture for monitoring the protocol conformance of a conversational Web service, the architecture facilitates the embedding for monitoring of other aspects of a service as well, such as QoS properties (i.e. response time), through plugging-in other monitors that may be more suitable for verifying particular properties.

Towards the aforementioned direction, it is necessary to classify the existing approaches for implementing a monitor. Thus, during the design of the monitoring architecture, we have identified two classes of monitors w.r.t their logical separation:

i) *Heavy-weight* monitor: A single monitor supports the monitoring of different aspects of a Web service. An intercepted request/response message is used for analysis for different monitored aspects within the same monitor, i.e. a monitor that is able to keep track of the responsiveness, availability and conformance of the conversational protocol of a service. Such a monitor depends upon a single session for a monitored service. However, this type is harder to implement, because it incorporates more complexity. Furthermore, a failure of the monitor implies inability to monitor any of the monitored aspects.

ii) *Light-weight* monitor: A single monitor supports the monitoring of one aspect of a Web service. Several light-weight monitors can be used to monitor diversified aspects of a service, i.e. three different monitors are required to monitor the same aforementioned properties. Although such monitors are not so complex to implement, the intercepted request/response messages need to be communicated to each monitor. Furthermore, every monitor has to preserve a session for the monitored service. However, the failure of a monitor has an impact only on the monitored aspect addressed by this monitor, thus the rest of the monitors are not affected.

Based on the aforementioned classification of the logical separation for monitor construction, we identify two solutions to enable dynamically plug-in and unplug monitors:

i) A single service acting as a message gateway for forwarding all requests/responses of the monitored services to specific monitors, which can be added and removed at run-time;

ii) A pool of different monitor services that are being attached to the monitored services at run-time.

The first approach offers a standardised way for intercepting messages. However, if a monitor needs more information, additional interception is necessary. Therefore, the message gateway service would have to be adjusted. Concerning the second approach, although monitors operate independently, the lack of a common gateway for intercepting request/response messages of the monitored services creates a barrier, since it will be necessary to configure the intercepting mechanism for each individual monitor. Hence, the first approach is less intrusive, because of a standardised way for intercepting, for reducing the necessary configuration overhead for monitoring a service.

The first approach was employed for the implementation of the monitoring architecture. The interception of request and response messages, together with the session management, logging and other joint functionality has been incorporated into a platform, which supports the deployment of multiple light-weight monitors that do not require to duplicate

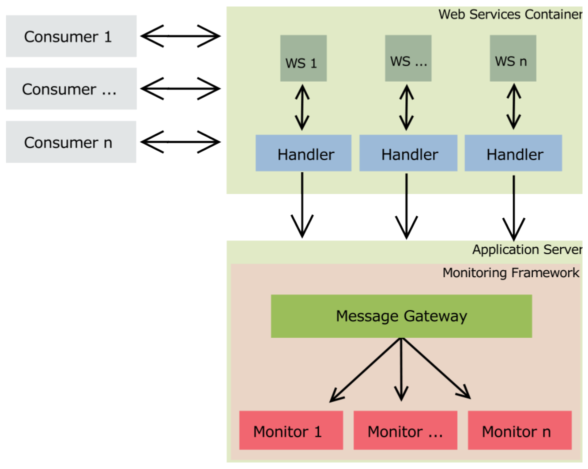

Figure 1: The components of the monitoring architecture.

features, since these features are already provided by the monitoring platform. Hence, the implementation of monitors becomes simpler, since each monitor has to address only the monitored aspect built for. This way only a simple interaction is associated with the monitored service, and thus facilitating less complexity regarding configuration overhead.

Another benefit of this approach is that a monitor can be deployed or suspended without interrupting the operation of the platform. However, it is necessary to bear in mind that although a fault occurring in a monitor may not affect other monitors, a fault occurring in the monitoring platform implies failure of all monitors, since every monitor depends upon the platform for its operation.

Figure 1 depicts the components of the architecture. Handlers, discussed in the next section, are used to intercept traffic from and to the Web services hosted in a Web services container. One or separate handlers can be attached to each Web service. The handler sends the intercepted messages to the monitoring framework, deployed on an application server, through the Message Gateway (MG). MG is a Web service that receives and forwards the messages, intercepted by handlers, to individual monitors, which are responsible for different monitoring aspects.

2.2 Message Interception

An important aspect of the monitoring architecture is the method used for intercepting the conversation, the exchanged request/response messages, between a service provider and a consumer. We consider three different approaches for message interception:

- *Handler-based Interception*: a handler is attached to the monitored service. The request/response messages are forwarded first to the handler, thus a handler is able to intercept them before reaching the monitored service and the consumer respectively.
- *Wrapper-Based Interception*: the monitored service is wrapped within another service. The resulting service has the same interface as the monitored service, and it delegates the messages to the monitored service. Thus, it is able to intercept the request/response messages exchanged between the monitored service and the consumer.
- *Proxy-based Interception*: an intermediate node acts as a network proxy. The proxy is able to intercept the request/response messages passing over the transport protocol, before they reach their destination.

Since the presented monitoring architecture uses the Message Gateway service for delegating intercepted messages to monitors, it can support all the three approaches. Although in the presented approaches the intercepted message is simply forwarded to the Message Gateway service, they have significant differences that affect the intrusiveness and the scalability of the monitoring architecture.

In wrapper-based interception, wrappers are independent of the underlying technology, in which the monitored service has been implemented, since they forward the request and response to and from the service. Hence, a wrapper appears as a consumer to the monitored service. Proxy-based interception is also independent of the technology, since it operates on the transport protocol of the communication. This is not the case however in handler-based interception, because the handler exists in the same Web service stack that exposes the monitored Web service. Thus, the handler has to be implemented in the same technology as the monitored service.

Another issue is the intrusiveness and transparency of the three approaches to the monitored service. The use of a wrapping service requires that all consumers change their binding to use the wrapped services. This is not necessary with a handler or a proxy, since both are transparent to the consumer.

If handlers are used, the monitored service will have to become unavailable during the deployment of a handler, because it would be necessary to change the configuration or even to recompile the Web service, depending on the underlying technology, in order the required handlers to operate properly. In contrast, the use of a wrapper or a proxy does not require the monitored service to become unavailable, because the wrapper is similar to any other consumer accessing the monitored service, and the proxy is transparent for the monitored service.

We have selected to explore the handler-based interception first, since handlers are widely supported by the available Web service technologies, and therefore require less effort to integrate with existing service-based applications. We recognise the importance of the other two approaches, wrapper and proxy based interception, which we plan to explore in future work.

2.3 Handling of Monitoring Sessions

Monitoring a conversational Web service that is being accessed concurrently from multiple consumers, requires the monitor to be able to determine for which session the request/response messages are processed. In order to address this issue, session handling facilities were implement in the monitoring framework.

For each conversation between the monitored service and a consumer, a new monitoring session is created. Every monitoring session is uniquely identified by a monitoring session identifier (MSID). The MSID has to be supplied during the interception and forward of a message to the Message Gateway service, in order for the monitoring framework to be able to identify the monitoring session that the forwarded message concerns.

2.4 Integration of JSXM Tool as a Monitor

We anticipate that the monitoring architecture presented in this paper will be of general use to different kinds of stakeholder involved in the provision and consumption of Web services, including service providers, service consumers, and service brokers, as an effective mean to monitor conversational services during their use. Therefore, the monitoring architecture was designed to be pluggable, such that it supports multiple concurrent monitors, which concern different monitoring aspects. Such an aspect is the behavioural conformance of the monitored service to its advertised specification during run-time.

In [8], we proposed a verification approach to run-time verification of behavioural conformance of Web services, which relies on the publication of a behavioural model for the Web service, based on the Stream X-Machines (SXM) [9] formalism described in the next section. The approach can be summarised as follows. A monitor simulates the SXM in parallel with the live service. The SXM model is animated with the use of the actual requests arriving at a Web service as inputs. Consequently, the expected responses are produced, which are then compared with the actual responses generated from the Web service.

The JSXM Tool [6] was integrated as a monitor in the implementation of the monitoring architecture. JSXM is a model-based testing tool implemented in Java, which is able to perform model animation, test generation and test transformation using an SXM model specified in an XML file. A portion of the functionality offered by JSXM is exposed through an API, in order to support animation of SXM models.

The intercepted request/response messages pass through a series of transformations, in order to feed the JSXM animator. The inputs are passed to the JSXM, which animates the SXM model and generates the outputs for the the given inputs. The inputs generated as a result of the model animation are compared with the transformed response messages of the monitored service. If the outputs match, the monitored service conforms to the SXM model, otherwise a deviation is detected.

In the aforementioned procedure, the JSXM tool serves as an oracle for forecasting the expected output of the monitored service. Mismatches are recorded in the monitoring log. Similarly, other verification tools could be incorporated as monitors that concern other monitoring aspects.

3. MODELLING CONVERSATIONAL WEB SERVICES AS SXMS

3.1 Stream X-machines

Stream X-Machines (SXMs) are special instances of the X-Machines introduced in 1974 by Samuel Eilenberg [9]. SXMs are a computational model capable of representing both the data and the control of a system. Although they utilise a diagrammatic approach of modelling control flow similar to the finite state machines, SXMs are capable of modelling non-trivial computation by embedding memory attached to the state machine. In addition, processing functions are used for representing transitions between states, instead of simple input symbols. Processing functions consume input symbols and read memory values, and generate output symbols while updating memory values. The introduction of the memory construct facilitates the reduction of the state explosion, since the number of states is reduced to critical states for the correct modelling of the system's abstract control structure. Some of the complexity is encapsulated in the transition functions, which can be later decomposed to simpler SXMs. This "divide-and-conquer" approach to design allows a top-down construction of the model.

A *(deterministic) SXM* [11] is defined as the tuple $(\Sigma, \Gamma, Q, M, \Phi, F, q_0, m_0)$ where:

- Σ and Γ are finite sets, called the *input alphabet* and the *output alphabet* respectively;
- Q is the finite set of *states*;
- M is a (possibly) infinite set called *memory*;
- Φ, which is called the *type* of the machine, is a finite set of partial functions (called *processing functions*) ϕ that map input-memory pairs to output-memory pairs, $\phi : \Sigma \times M \to \Gamma \times M$;
- F is the next state partial function that given a state and a processing function from the type Φ, provides the next state, $F : Q \times \Phi \to Q$;
- q_0 and m_0 are the *initial state* and *initial memory* respectively.

The most significant advantage of SXMs is a testing method [11], which allows the verification of the conformance of a system's implementation against its specification. Under the test hypothesis that the system is made of fault-free components and the satisfaction of some well defined design-for-test conditions [11], it is ensured that the system behaviour is functionally identical to that of the implementation, if the application of a finite test set, which is generated by the aforementioned method, produces the same results both in the specification and the implementation.

For testing purposes the model is used both for test generation (producing the inputs to test against) and as an oracle for providing the expected outputs. For the purpose of runtime monitoring the model is only needed to be used as an oracle, since the inputs are provided by the monitored system.

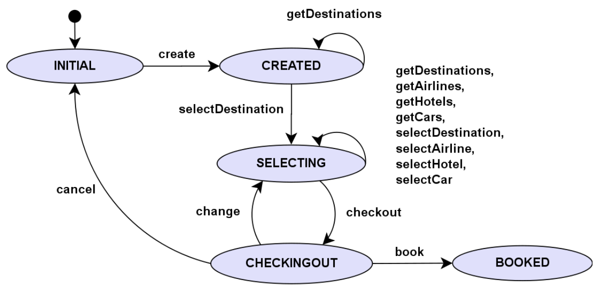

Figure 2: The state-transition diagram (associated automaton) of a SXM representing the TravelAgency service.

3.2 SXM Models for Web services

Conversational Web services can be easily modelled as SXMs, since they both consume inputs and generate outputs, while they modify their internal state.

The construction of a SXM model for a Web service requires that SXM inputs and outputs are created from the SOAP request and SOAP response messages of the service respectively. The transition functions of a SXM are specified based on the Web service operations. Although the transitions are derived directly from the Web service operations, each operation may map to one or more transitions (processing functions) dealing with different branches of the computation that depend on state (and indirectly on inputs). For more detailed explanation about modelling Web services as SXMs refer to [7, 12].

4. VALIDATION

4.1 The Web Service example

During the implementation of the monitoring architecture, several examples have been constructed for the evaluation of the implemented architecture. In this paper we present the TravelAgency, a service that offers functionality for booking complete travel packages. The TravelAgency service is an example of a private-transient state conversational service, since it has been implemented in such a way that its local state cannot be modified by an external component, and its state is initialised at the beginning of each session and destroyed upon completion.

The described service utilises transport-related session management techniques to manage concurrent consumers, by relying on the underlying HTTP session in order to associate the state information for each consumer accessing the service. Thus, the travel package prepared by a consumer is stored in the HTTP session. For convenience, we use directly the HTTP session token as the monitoring session identifier (MSID). We would like to stress that the service has been implemented in such a way in order to achieve a conversational behaviour. In a real system, the TravelAgency service will be a stateless interface which provides access to the operations of a complete booking system.

Figure 2 illustrates the associated state transition diagram of the SXM model for the TravelAgency service. The service operates as follows: First, a travel package is created that will store all the travelling preferences of the consumer. Next, different travelling destinations can be fetched in order to choose a preferred destination. After a destination has been selected, the available airlines, hotels, and cars can be fetched in order to customise the travel package. In order to checkout, an airline and a hotel need to be selected. Optionally the consumer can also select a car. At this stage, the travel package can be cancelled, modified or booked. Booking the travel package ends the interaction with the service.

Each processing function, presented in Figure 2, has been specified in the complete SXM specification of the TravelAgency service, using the approach that has been presented in [8]. The complete SXM specification is beyond the scope of this paper and thus is not included.

4.2 Evaluation Setup

An evaluation of the monitoring architecture was performed using the TravelAgency service. The aim of this experiment was to evaluate scalability of the monitoring architecture w.r.t performance.

The evaluation scenario involves many consumers accessing the TravelAgency service concurrently. Multiple executions were done with the monitor enabled and disabled, and with an increasing number of concurrent consumers. The TravelAgency and the consumers were hosted by different computers located in the same local network, so that the measurements would not be affected by long network delays.

Table 1 presents four interaction scenarios. Each scenario contains different sequences of operation invocations for the TravelAgency service, in order for it to be possible to observe different behaviours. For instance, scenario 1 books a travel package successfully, therefore it completes successfully the interaction with the TravelAgency service, whereas scenario 4 fails to book a travel package, because it attempts to invoke the book() operation, without first invoking the checkout() operation.

Table 1: Interaction scenarios containing different execution sequences.

Scenario	Effect	Execution
1	Booked	Succeeds
2	Booked with car	Succeeds
3	No hotel is selected	Fails
4	No checkout is done	Fails

The TravelAgency service has been implemented as a stateless session Enterprise Java Bean (EJB), which has been exposed as a Web service using JAX-WS annotations [3]. A handler has been attached to the service, in order to intercept request/response message and then forward them to the Message Gateway.

JBoss Application Server (JBoss AS) [2] was used for the deployment of the involved components. JBoss AS embeds a JAX-WS Web service stack with the use of JBoss WS. The hardware configuration of the server machine, used for hosting JBoss AS, was an Intel Core 2 Quad Q8400 with 4GB of ram at 1066MHz running openSUSE 11.2. Both the TravelAgency service and the monitoring framework were

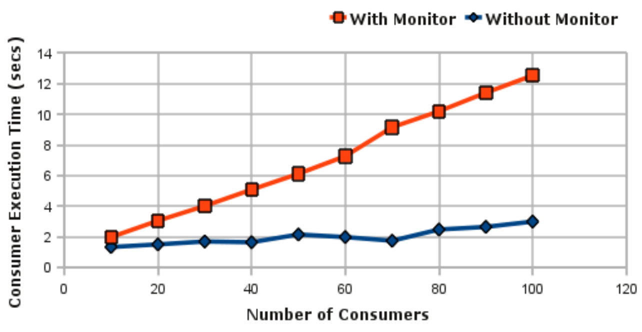

Figure 3: Average execution time per consumer complete interaction for each test run.

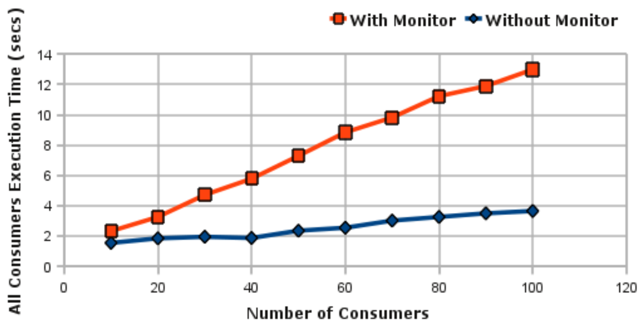

Figure 4: Average total execution time for all consumer complete interaction for each test run.

deployed in the same JBoss AS, resulting in provider-based monitoring scenario.

In order to produce more accurate and realistic results, we used profiling techniques that utilise dynamic analysis of a program, The Eclipse Test and Performance Tools Platform (TPTP) [1] was used to perform execution time analysis and thread analysis of the consumers accessing the service.

4.3 Experimental Results

A series of test runs was carried out, by instantiating multiple consumers, which were accessing the TravelAgency service concurrently. Before initiating a test run, a restart of the JBoss AS was performed, in order to avoid the test run execution being affected from previous trials. In addition, each test run was performed three times and the average measurement was considered, in order to produce more accurate results.

Ten test runs were carried out during the experimentation phase. In each test run the consumer population was growing linearly, in order to simulate increased traffic on the TravelAgency service. The consumer population consisted of all four consumer types. In the first test run only 10 consumers were instantiated, while in every following test run another 10 consumers were added, resulting in a total of 100 consumers at the final test run.

Instead of measuring the overhead that the monitoring introduces in a single invocation of a service method, we decided to measure the overhead at the interaction level, in order to produce a more realistic evaluation for the impact of the monitoring activities during the operation of the service. Thus, for each test run two measurements were performed. First, the time required for a consumer to complete the interaction with the service was measured. Second, the total time required for all consumers to complete their interaction with the service was measured. Both measurements were executed once while the service was not being monitored, and once with the service being monitored.

Figure 3 shows the recorded measurements concerning the execution time required for a consumer to complete the interaction with the TravelAgency service, with and without the monitor. The execution time of each consumer linearly increases as the consumer population increases.

Figure 4 depicts the recorded measurements regarding the execution time required for all consumers to complete their interaction with the TravelAgency service, with and without the monitor. The total execution time increases linearly also. The comparison of Figure 4 to Figure 3 reveals that even if all consumers begin their execution almost simultaneously, they also complete their execution at the same time.

The measurements presented so far demonstrate that there is a noticeable increase in the time required for concurrent consumers to complete their execution. Although the overhead appears to be significant, it worth noticing that the monitor achieves to handle all monitored services in a timely fashion. The overhead was generated due to thread synchronisation within the handlers used to intercept request/response messages. The thread synchronisation caused the consumers threads to suspend at different points of the execution.

4.4 Error Discovery

Within the scope of the experiment, we observed the ability of the monitor to detect errors. We evaluated the error discovery ability of the JSXM monitor w.r.t. behavioural conformance of the monitored service to its specification.

The first phase of the evaluation concerned consumers following a different interaction protocol (e.g. ClientBookNoCheckout), which was not supported by the specification of the TravelAgency service. In the next phase, we used consumers following the interaction protocol described in the specification. However, we altered the implementation of the TravelAgency service, so that it follows a different interaction protocol. For instance, allowing it to perform checkout without having selected a hotel, which is different from the specification that requires both an airline and a hotel to be chosen before continuing to checkout. In both cases the monitor was successful in detecting the errors in the interaction protocol.

Currently, the monitoring framework lacks the ability to detect lost (dropped) SOAP request/response messages. The lack of this knowledge may lead to false conclusions that a service is behaving faulty. A more sophisticated diagnosis approach based on the aggregated observations from different monitors (e.g. timeout monitors) would produce joint conclusions explaining the cause of the mismatch.

5. DISCUSSION

The monitoring architecture is designed to support effortless integration in a service-based application. The monitor can be deployed on the same or a different application server from the monitored services, and thus execute independently. Additionally, the monitor is compatible with any underlying technology that supports Web services, since it is agnostic of the particular programming languages used for implementing the actual service.

The three presented message interception approaches, handler-based, wrapper-based, and proxy-based interception can be used within the monitoring architecture, thus, the integration of the monitoring activity into SOA infrastructures appears easier. Also, the separation of the interception mechanism from the monitors, allows the monitors to be hosted by a third-party trusted by both the service provider and the service consumer for verifying properties of Web services under monitor.

At the current state, we have integrated a monitor (JSXM) which concerns the monitoring of the functional aspect of a conversational Web service. The integration of a monitor that concerns the non-functional aspect could reveal new requirements for the presented monitoring architecture. Therefore We plan to investigate the integration of non-functional monitors in further work.

The experimental results have indicated a linear increase in execution time that depends on the number of concurrent consumers. It is worth noticing that in a production environment interaction is often driven by people, and therefore it is not realistic that 100 users would complete a 5-turn transaction in 12 seconds. Nevertheless, in order to improve the efficiency of monitoring the monitors could execute asynchronously from the monitored service. This solution which would introduce concurrency and thread synchronisation issues is going to be investigated in future work.

The implementation of long-running transactions requires the use of persistent-state services, which are able to store and restore the state of the consumer session, so that a consumer can interrupt service usage and continue at a later point in time. In order for the monitoring architecture to support persistent-state services, it is necessary that the state of each monitor for the monitored service is stored. Furthermore, additional identification information is needed, since the underlying transport session will expire, and thus client identification will not be possible. In addition, maintaining concurrent monitoring sessions for multiple persistent-state services is resource intensive, and it is therefore necessary to use caching techniques for the monitoring sessions, in order to be able to suspend and resume a monitoring session.

The JSXM monitor needs to convert the actual request and response messages to concrete inputs and outputs compatible with the SXM specification. This means that the request/response messages need to pass a transformation, in order to be lifted (abstracted) to a usable representation for the JSXM monitor. We believe that the need for abstracting or transforming request/response messages would concern other types of monitors that could be implemented in the future. Hence, information could be provided to the monitor in order to perform the necessary transformations. These information could be either provided as a configuration in the monitor, or published together with the Web service description using SAWSDL. Towards similar direction in [14], we have presented an algorithm to convert inputs, outputs, preconditions and effects (IOPE), published through SAWSDL, for constructing a stateful EFSM specification of a service for verification.

6. RELATED WORK

Significant effort has been directed toward the creation of a viable monitoring framework for Web services. Different methodologies have been developed, in order to provide monitoring facilities for the functional and the non-functional aspects of Web services.

Li et al [13] proposed a framework for monitoring run-time interaction behaviour of Web services. Validation of predefined interaction constraints is performed using finite state automata. Our work differs since we attempt to support multiple monitoring techniques under a common framework. Furthermore, the message interception that they employ is bound to the particular server used for deploying Web services. The presented monitoring architecture supports message interception independent of the underlying infrastructure.

Zulkernine et al [17] proposed a framework for performance monitoring of Web services, which is part of a greater middleware solution. They also use handlers to intercept messages and measure the responsiveness of a service. They present an evaluation of the framework where they measure the overhead of the monitor w.r.t. the response time of the service. However, their evaluation is limited to 10 concurrent consumers only, and it appears that the overhead introduced is significantly greater than in our approach in some cases. The overhead reported in that work increases at approximately twice the rate as in our approach.

Simmonds et al [15] proposed a more complete monitoring framework for checking behavioural correctness of Web service conversations. They use UML 2.0 Sequence Diagrams as a property specification language, which are then transformed to automata by multiple monitors that check the validity of safety and liveness properties. They intercept messages with the use of handlers. Although they have implemented a similar approach to interception and handling of messages, their proposal appears to be specific to the used application server, since they utilise an event mechanism provided by that server.

Alodib and Bordbar [4] proposed an approach for monitoring by employing Workflow Graphs as the underlying specification language, for generating monitors that are exposed as Web services. However, they do not suggest a unified architecture of monitors, but rather a methodology for deriving and using individual monitors. The experimental results reported are similar to the results of our work.

Guinea et al [10] proposed a monitoring framework that integrates three different monitoring approaches. The framework is able to report and monitor functional requirements and quality of service constraints for BPEL processes. This approach leverages data collection, including message interception. for monitoring. Although this work aligns with our objectives w.r.t. the extensibility of the architecture, it appears to be more applicable for orchestrator-based monitoring, since the approach requires internal inspection of the monitored process.

Wetzstein et al [16] proposed an approach to event-based monitoring of process metrics across participants in a choreography. Their work is related to Business Activity Monitoring (BAM) within Service-Oriented Architectures (SOA). The authors use a monitoring agreement written in XML to specify what should be monitored. Our monitoring architecture is less intrusive, since it does not require any modification to the execution container as the one presented in [16].

In contrast with the presented related work, the main contribution of this paper is not the particular monitoring tool (JSXM), but a monitoring architecture that can be used for integrating different monitoring approaches or tools. We envision the presented monitoring architecture as a platform for deploying several monitoring tools, which monitor the functional as well as the non-functional aspects of conversational Web services.

7. CONCLUSION

Monitoring conversational Web services involves a variety of issues such as message interception, session management, and concurrency. In order to support the integration of different monitoring approaches, it is important to create a holistic monitoring architecture, which offers the common facilities required for monitoring of Web services, such as message interception and session handling.

This work presented an extensible architecture for run-time monitoring of conversational Web services. The architecture has been designed and implemented to facilitate integration with the existing service-oriented architectures, and to allow the use of different monitoring approaches. The experimental results revealed that the monitoring activities have a tolerable overhead on the operation of a Web service.

As future work the efficiency of monitoring will be improved by resolving concurrency issues. Moreover, we will expand the monitoring framework to support persistent-state services. The wrapper-based and proxy-based interception approaches will be implemented and evaluated. Finally, we plan to investigate how the presented monitoring architecture could be extended to a generic framework supporting the integration of monitors for both non-functional and functional aspects of conversational as well as non-conversational Web services.

8. REFERENCES

[1] Eclipse test & performance tools platform project, http://www.eclipse.org/tptp/.

[2] Jboss application server, http://www.jboss.org/jbossas.

[3] JSR 224: JavaTM API for XML-Based web services (JAX-WS) 2.0. Technical report, Java Community Process, http://jcp.org/en/jsr/detail?id=224, 2009.

[4] M. Alodib and B. Bordbar. A Model-Based approach to fault diagnosis in service oriented architectures. In *2009 Seventh IEEE European Conference on Web Services*, pages 129–138. IEEE, 2009.

[5] L. Baresi, S. Guinea, M. Pistore, and M. Trainotti. Dynamo + astro: An integrated approach for BPEL monitoring. In *Web Services, IEEE International Conference on*, volume 0, pages 230–237, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[6] D. Dranidis. JSXM: A Suite of Tools for Model-Based Automated Test Generation: User Manual. Technical report, Technical Report WPCS01-09, CITY College, 2009, 2009.

[7] D. Dranidis, D. Kourtesis, and E. Ramollari. Formal verification of web service behavioural conformance through testing. *Annals of Mathematics, Computing & Teleinformatics*, 1(5):36–43, 2007.

[8] D. Dranidis, E. Ramollari, and D. Kourtesis. Run-time Verification of Behavioural Conformance for Conversational Web Services. In *2009 Seventh IEEE European Conference on Web Services*, pages 139–147. IEEE, 2009.

[9] S. Eilenberg. Automata, languages and machines. *Academic Press, New York*, A, 1974.

[10] S. Guinea, L. Baresi, G. Spanoudakis, and O. Nano. Comprehensive monitoring of BPEL processes. *IEEE Internet Computing*, Nov. 2009.

[11] M. Holcombe and F. Ipate. *Correct Systems: Building Business Process Solutions.* Springer Verlag, Berlin, 1998.

[12] D. Kourtesis, E. Ramollari, D. Dranidis, and I. Paraskakis. *Discovery and Selection of Certified Web Services Through Registry-Based Testing and Verification.* IFIP International Federation for Information Processing, Pervasive Collaborative Networks. Springer, 2008.

[13] Z. Li, Y. Jin, and J. Han. A runtime monitoring and validation framework for web service interactions. In *Proceedings of the Australian Software Engineering Conference*, pages 70–79. IEEE Computer Society, 2006.

[14] E. Ramollari, D. Kourtesis, D. Dranidis, and A. Simons. Leveraging Semantic Web Service Descriptions for Validation by Automated Functional Testing. *The Semantic Web: Research and Applications*, pages 593–607, 2009.

[15] J. Simmonds, Y. Gan, M. Chechik, S. Nejati, B. O'Farrell, E. Litani, and J. Waterhouse. Runtime monitoring of web service conversations. *IEEE Transactions on Services Computing*, 99(1):223–244, 2009.

[16] B. Wetzstein, D. Karastoyanova, O. Kopp, F. Leymann, and D. Zwink. Cross-organizational process monitoring based on service choreographies. In *Proceedings of the 2010 ACM Symposium on Applied Computing - SAC '10*, page 2485, Sierre, Switzerland, 2010.

[17] F. H. Zulkernine, P. Martin, and K. Wilson. A middleware solution to monitoring composite web Services-Based processes. In *Proceedings of the 2008 IEEE Congress on Services Part II*, pages 149–156. IEEE Computer Society, 2008.