
Proceedings Paper:
Dranidis, Dimitris, Ramollari, Ervin and Kourtesis, Dimitrios (2009) Run-time verification of behavioural conformance for conversational web services. In: 2009 Seventh IEEE European Conference on Web Services. 2009 Seventh IEEE European Conference on Web Services, 09-11 November 2009, Eindhoven, The Netherlands. IEEE Computer Society , pp. 139-147. ISBN 978-0-7695-3854-9

https://doi.org/10.1109/ECOWS.2009.19

2009 Seventh IEEE European Conference on Web Services

Run-time Verification of Behavioural Conformance for Conversational Web Services

Dimitris Dranidis
Computer Science Department, CITY College
Affiliated Institution of the University of Sheffield
Thessaloniki, Greece
dranidis@city.academic.gr

Ervin Ramollari, Dimitrios Kourtesis
South East European Research Centre (SEERC)
Research Centre of the University of Sheffield and
CITY College
Thessaloniki, Greece
erramollari@seerc.org, dkourtesis@seerc.org

Abstract— **Web services exposing run-time behaviour that deviates from their behavioural specifications represent a major threat to the sustainability of a service-oriented ecosystem. It is therefore critical to verify the behavioural conformance of services during run-time. This paper discusses a novel approach for run-time verification of Web services. It proposes the utilisation of Stream X-machines for constructing formal behavioural specifications of Web services which can be exploited for verifying that a service's run-time behaviour does not deviate from what is defined in the specification. Our approach allows for checking both the control flow of a Web service and the values of the data in the generated responses. The paper also proposes a classification of Web services and discusses how different types of services can be verified at run-time. Finally, it presents a run-time monitoring and verification architecture and discusses how it can be integrated into different types of service-oriented infrastructures.**

Keywords-Run-time verification; behavioural conformace; Web services; monitoring; formal methods; Stream X-machines

I. INTRODUCTION

Web services are typically accompanied by some form of technical specifications that explicate various functional or non-functional aspects of a service's operation. The most elementary and well-known example of such technical specifications is the description of a SOAP Web service's interface that is encoded using WSDL (Web Service Description Language). The specifications of a Web service are indispensable for carrying out a variety of activities throughout the service's lifecycle, such as discovery and composition, and the degree of formality in the specifications determines the extent to which these activities can be automated.

Often, however, the behavioural specification of a service and its implementation may prove to be inconsistent. Even in cases where a Web service has been demonstrated to conform to its specification before deployment, it is possible that the service is eventually found to deviate from its specification at run-time. This can happen for several reasons. One reason could be that the service provider has inadvertently or intentionally modified the Web service implementation at some point after its initial deployment without reflecting this change in the specification. Another reason could be that the Web service is composite and depends on other services which eventually become unavailable, get replaced, or have their behaviour modified. In all these cases the Web service's functionality can become affected in a way that prevents it from conforming to its original specification.

The introduction of such "defective" Web services within a highly dynamic service-oriented environment can have a devastating effect that puts the sustainability of the whole infrastructure at risk. Therefore, being able to ensure that the run-time behaviour of services does not deviate from their advertised specifications is considered critical.

Run-time verification is a verification technique which is based on program execution and ensures that the program conforms to its intended behaviour at run-time. Service run-time verification consists of observing and logging the input/output traces of service execution and verifying that the observed behaviour satisfies given requirements. The intended behaviour of the service needs to be described in the service specification and be expressed in some formal language.

Previous work by the authors has focused on specifying the behaviour of stateful and conversational Web services using Stream X-machines (SXMs) [15, 11], which are computational models constituting a type of Extended Finite State Machines. SXM models are utilised for the automated generation of exhaustive test cases, which, under well defined conditions, can guarantee to reveal all inconsistencies among the implementation of a Web service and its expected behaviour [6]. In [19] an approach is proposed for reliable Web service discovery through behavioural verification and validation based on SXMs. In [13] and [14] the authors put forward the development of an extended service registry which receives the SXM specification of a Web service upon publication, and subsequently performs test generation and test execution in order to verify that the service implementation conforms to its specification. In [20], an approach is proposed for specifying the behaviour of services in terms of inputs, outputs, preconditions, and effects utilising ontology-based (OWL-DL) and rule-based descriptions (RIF-PRD), and deriving SXM specifications for functional testing and validation.

In this paper we propose a novel approach for verifying the conformance of conversational Web services at run-time. The novelty of this approach lies in the utilisation of Stream X-machines for specifying expected service behaviour and guiding the monitoring process through SXM model animation. To the best of our knowledge this

978-0-7695-3854-9/09 $26.00 © 2009 IEEE
DOI 10.1109/ECOWS.2009.19

is the first work using some form of Extended Finite State Machines for run-time verification of Web services. In contrast with other approaches which are discussed in the related work, SXM modelling does not only allow the checking of the control-flow of operation invocations (protocol checking) but also verifies the correct implementation of the operations (functional conformance). Furthermore, we believe that the proposed state-based formalism is more intuitive to the software engineer for the description of stateful services than other formalisms utilised in other approaches. Additionally, we suggest a classification of stateful Web services, which allows us to identify different types of services for which we can perform monitoring and run-time verification. We also propose a conceptual architecture for the implementation of run-time monitoring and verification infrastructure, and discuss how it is integrated into different scenarios of Service-Oriented Architecture (SOA) configurations involving the service consumer, provider, and broker. Compared to other Web service run-time monitoring approaches the proposed solution allows the integration of the monitoring architecture both at the consumer's and the provider's site, as well as at the broker's site.

The rest of the paper is structured as follows. In section 2 we present a classification of Web services for the purposes of run-time verification. Section 3 provides an overview of the SXM modelling formalism along with an example for the purposes of illustration. Section 4 describes in detail how run-time verification is performed for different types of services. Section 5 discusses the integration of run-time monitoring into different SOA infrastructures. Finally, in section 6 we provide an overview of some related work, and in the conclusions we summarise the main points from our work and provide an outlook for future research.

II. CLASSIFICATION OF WEB SERVICES

Web Services can be generally distinguished among *stateless* and *stateful* ones. By the term stateless, we refer to services that do not exhibit any observable state. In a stateless service the response of any operation depends solely on the provided input arguments; the same result is delivered for the same arguments every time the operation is invoked (e.g. in some service offering mathematical calculations). The functional behaviour of such an operation can be specified by a contract, in terms of pre-conditions on operation inputs and post-conditions on operation outputs, and its run-time behaviour can be monitored in isolation from other operations. In contrast, in a stateful service, the response of an operation depends not only on the input arguments but also on the internal state of the service. This is why pre-conditions on operation inputs and post-conditions on operation outputs do not suffice for monitoring the observable behaviour of a stateful service. This work focuses on the run-time verification of stateful services, for which we propose three further classifications.

Firstly, stateful services can be distinguished into *conversational* and *non-conversational* services:

- In a *non-conversational* service all operations can be accepted (terminate successfully without producing any error) at all states. This means that all operations will return some result when invoked, independently of the state in which the service is found at the time of invocation.
- In a *conversational* service only specific operation sequences are accepted. This is because some operations may have preconditions which depend on the state of the service. The set of all acceptable operation sequences is called the protocol or choreography of the interaction.

A shopping cart service is a typical example of a stateful and conversational service. Items cannot be removed from the cart if they were not added in the cart in a previous step. A currency converter service and a random number generator consisting of a single "getNextRandomNumber" operation are examples of stateful and non-conversational services: their operations will return different results for the same inputs at different times of invocation, but no particular protocol is imposed.

Secondly, we differentiate among cases where the state of a service-instance (spawned to serve a particular client during a session) can be modified only by that specific service-instance, or by other service-instances and external systems. Therefore, stateful services can be further categorised depending on the type of state modifiability to:

- *Private-state* services: the state of each service instance depends exclusively upon and is fully determined by the sequence of previous operation invocations. Thus, the behaviour of the service depends only on the current local state resulting exclusively from the interactions among the client and the service-instance.
- *Shared-state* services: the state of the service cannot be fully determined by the sequence of service invocations. The behaviour of the service instance depends on some state variables which may be modified by other service instances or applications.

The previously mentioned non-conversational random number generator service is a private-state service, since the value to be returned whenever an operation is invoked is determined fully and solely by the history of previous invocations. In contrast, the non-conversational currency converter service is a shared-state service, since the value to be returned upon invocation is determined by the state of externally controlled and dynamic exchange rates.

A conversational shopping cart service can be a private-state service or a shared-state service, depending on its implementation. A shopping cart service whose behaviour is affected only by the state of the internal shopping basket information is a private-state service. In contrast, a shopping cart service whose behaviour also depends on externally-controlled inventory information is a shared-state service. In the first case, the addition of an item to the cart would always be successful, unlike the second case, where trying to add an item to the cart could fail if the item was not available in stock or unknown in the inventory list.

It should be noted that from the observer's point of view the behaviour of a shared-state service is always non-deterministic. This is because the actual result of an operation cannot be known in advance. Therefore, the formalism employed for modelling the behaviour of such services and the mechanism used for monitoring them

should allow for non-determinism. A non-deterministic model of a service should be able to describe the set of all possible permitted behaviours and responses. The implications for supporting non-determinism in the monitoring mechanism are examined in section IV.

TABLE I. EXAMPLES OF STATEFUL SERVICES.

	Non-conversational	Conversational
Private-state	**Random number generator**	**Shopping cart without inventory lookup**
Shared-state	**Currency converter**	**Shopping cart with inventory lookup**

In [3] a slightly different classification is used, naming the private-state and shared-state services as conversational and global shared resource services, respectively. In our classification, however, shared state services can also be conversational. Our classification of private-state versus shared-state services closely matches with the private-data-driven versus public-data-driven Web service categories described in [4].

Thirdly, we propose distinguishing among services that operate on:

- *Transient-state*: the state information is volatile, i.e. initialised at the beginning of each session of a service-instance and destroyed upon completion.
- *Persistent-state*: The state information outlasts the duration of a session. The service "remembers" its state even after the end of a session. Different service instances from the same client can thus interrupt service usage and continue at a later point in time.

An example of a transient-state service is a shopping cart service in which the client always begins with an empty shopping cart, whereas a persistent-state shopping cart service "remembers" old additions in the shopping cart. In the latter case, the client needs to identify the shopping cart by providing its id as part of the operation arguments.

Note that persistent-state services require some way of client identification so that a newly created service-instance can restore its saved state for the specific client. This is also important for run-time monitoring. Monitors should be able to save their state and restore it at a later point of time in order to be able to continue monitoring.

From the classification presented in this section it becomes clear that different types of Web services require different treatment for the purpose of run-time verification. In this paper we mainly focus on the run-time monitoring of conversational services, i.e. those services which follow a protocol. These can be either private-state or shared-state and either transient-state or persistent-state. In section IV we present how these different types are handled in our monitoring framework.

III. MODELLING WEB SERVICES WITH SXMS

A. Stream X-machines

Stream X-machines (SXMs) are a computational model capable of representing both the data and the control of a system. SXMs are special instances of the X-machines introduced in 1974 by Samuel Eilenberg [8]. They employ a diagrammatic approach of modelling control flow by extending the expressive power of finite state machines. In contrast to finite state machines, SXMs are capable of modelling non-trivial data structures by employing a memory attached to the state machine. Moreover, transitions between states are not labelled with simple input symbols but with processing functions. Processing functions receive input symbols and read memory values, and produce output symbols while modifying memory values. Adding the memory construct allows the model designer to reduce the number of states to those states which are considered critical for the correct behavioural verification of the system; in other words the states and transitions that are modelled are those that need to be verified. A divide-and-conquer approach to design allows the model to hide some of the complexity in the transition functions, which can be later exposed as simpler SXMs at the next level.

A Stream X-machine is defined as an 8-tuple, $(\Sigma, \Gamma, Q, M, \Phi, F, q_0, m_0)$ where:

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

Apart from being formal as well as proven to possess the computational power of Turing machines, SXMs offer a highly effective testing method for verifying the conformance of a system's implementation against a specification [11]. This method generates test sequences whose application ensures that the system behaviour is identical to that of the implementation, provided that the system is made of fault-free components and some explicit design-for-test conditions (controllability and observability) are met. Although testing can increase the confidence that the system is behaving in the right way it cannot guarantee fault free operation during run-time, especially in the case of a highly dynamic environment as that of Web services.

B. Supporting tools

Stream X-machine models can be represented in a number of languages. XMDL (X-Machine Definition Language) is a special-purpose mark-up language introduced in [12]. XMDL-O is an object based extension of XMDL introduced in [5]. Both languages are supported by a Prolog-based tool named X-system. The X-system facilitates the modelling of SXMs by providing a parser and an animator.

Recently a new suite of supporting tools has been developed in Java (JSXM) to support automated model-based test generation [7]. The JSXM tool introduces a new syntax for SXM specifications based on XML and Java

inline code. The XML-based specifications in JSXM facilitate easier integration with Web technologies and related XML-based Web service standards.

JSXM supports animation of SXM models, model-based test generation and test transformation. The test generation is based on the SXM testing theory and given an SXM model it generates a set of test cases in XML format. As such, the test cases are independent of the programming language of the implementation. Test transformation is used for transforming the general test cases to concrete test cases in the underlying technology of the implementation. Currently, a Java test transformer is available which automatically generates JUnit test cases.

The animator tool enables the execution (or simulation) of SXM models. This is the main part of our monitoring architecture which is presented in section IV. Once the animator is launched, it initialises the SXM model and then repeatedly accepts an input, feeds the input to the SXM model, and retrieves the produced output. In that sense the animator can serve as an oracle that provides the expected outputs for the purpose of run-time monitoring. The animator can be executed in interactive mode or in batch mode by receiving an XML file of the inputs. An API is also provided which allows calling the animator's methods programmatically. This API is utilised in the implementation of the run-time verification architecture described in section IV.

C. Modelling Web services

In order to model the behaviour of a Web service using a Stream X-machine, the modeller must perform data-level and behaviour-level analysis to derive the appropriate SXM modelling constructs. Parallels can be drawn between a stateful Web service and a Stream X-machine, since they both accept inputs and produce outputs, while moving from one internal state to another. SXM inputs correspond to SOAP request messages, outputs correspond to SOAP response messages, and processing functions correspond to Web service operation invocations in specific contexts (an operation may map to more than one processing functions because of the potentially different input parameters and the different state). SXM-based modelling is applicable in the context of complex conversational Web services where the result obtained from invoking a Web service operation depends not only on the consumer's input, but also on the internal state of the service.

Fig. 1 illustrates the state transition diagram from an SXM model of a stateful Web service that allows performing basic operations on a bank account. This service is another example of a private-persistent state conversational service. For simplicity, let us assume that the Web service interface comprises five operations: open, deposit, withdraw, getBalance, and close. The service normally manages several bank accounts, so each operation takes an account ID as a parameter. However the model is abstracted by focusing on a single account instance, identified by the same ID in all operations. This modelling approach is based on the Manager Pattern described in [2]. When an account is created it is initialised as inactive and therefore needs to be set to active (opened) before any transaction can be performed. The deposit of an amount results in increasing the balance of the account as appropriate, while the withdrawal of an amount can take place only if the amount does not exceed the balance, and results in reducing the balance accordingly. Finally, an account can be closed only if its balance is zero, and once closed it cannot be re-activated.

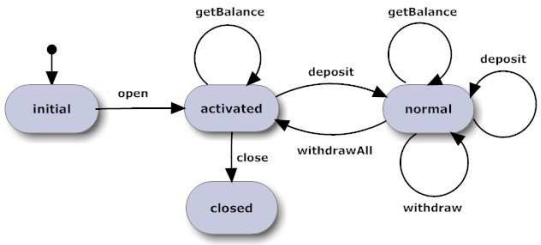

Figure 1. A state transition diagram for a Stream X-machine modelling a bank account Web service.

The corresponding code for representing the state transition diagram in JSXM is provided below:

```
<SXM name="Account">
  <states>
    <state name="initial" />
    <state name="activated" />
    <state name="closed" />
    <state name="normal" />
  </states>
  <initialState state="initial" />

  <transitions>
    <transition from="initial" function="open" to="activated" />
    <transition from="activated" function="close" to="closed" />
    <transition from="activated" function="deposit" to="normal" />
    <transition from="normal" function="deposit" to="normal" />
    <transition from="normal" function="withdrawAll" to="normal" />
    <transition from="normal" function="withdraw" to="activated" />
    <transition from="activated" function="getBalance" to="activated"/>
    <transition from="normal" function="getBalance" to="normal" />
  </transitions>
```

The memory of the account SXM just stores the current balance of the account. In the following the integer memory variable balance is declared and initialised:

```
  <memory>
    <declaration>
       int balance
    </declaration>
    <initial>
       balance = 0
    </initial>
  </memory>
```

The inputs correspond to SOAP request messages and may consist of arguments:

```
  <inputs>
    <input name="openRequest" />
    <input name="closeRequest" />
    <input name="getBalanceRequest" />
    <input name="depositRequest">
       <arg name="amount" type="Int" />
    </input>
    <input name="withdrawRequest">
       <arg name="amount" type="Int"/>
    </input>
  </inputs>
```

The outputs correspond to SOAP response messages and are structured in a similar way to inputs:

```
<outputs>
  <output name="openResponse" />
  <output name="closeResponse" />
  <output name="depositResponse">
    <result name="amount" type="Int" />
  </output>
  <output name="withdrawResponse">
    <result name="amount" type="Int" />
  </output>
  <output name="getBalanceResponse">
    <result name="amount" type="Int" />
  </output>
</outputs>
```

Processing functions are specified by defining their inputs, outputs, preconditions and effects on the SXM memory. Note that two processing functions may be triggered by withdrawRequest messages. However, their complementary guard conditions make sure that they cannot be triggered at the same time (deterministic choice).

```
<functions>
  <function name="open" input="openRequest"
                        output="openResponse"/>

  <function name="close" input="closeRequest"
                        output="closeResponse" />

  <function name="deposit" input="depositRequest"
                        output="depositResponse">
    <precondition>
      depositRequest.amount > 0
    </precondition>
    <effect>
      balance = balance + depositRequest.amount;
      depositResponse.amount = depositRequest.amount;
    </effect>
  </function>

  <function name="withdraw" input="withdrawRequest"
              output="withdrawResponse">
    <precondition>
      withdrawRequest.amount > 0 &&
      balance > withdrawRequest.amount
    </precondition>
    <effect>
      balance = balance - withdrawRequest.amount;
      withdrawResponse.amount = withdrawRequest.amount;
    </effect>
  </function>

  <function name="withdrawAll" input="withdrawRequest"
              output="withdrawResponse">
    <precondition>
      withdrawRequest.amount > 0 &&
      balance == withdrawRequest.amount
    </precondition>
    <effect>
      balance = balance - withdrawRequest.amount;
      withdrawResponse.amount = withdrawRequest.amount;
    </effect>
  </function>

  <function name="getBalance" input="getBalanceRequest"
              output="getBalanceResponse">
    <effect>
      getBalanceResponse.amount = balance;
    </effect>
  </function>
</functions>
```

The process of deriving the SXM specification of a Web service is usually manual. However, in [20] a method is presented to semi-automatically derive an SXM specification from a semantically annotated Web service utilising ontology-based and rule-based descriptions.

Once completed, the SXM model can be used for both verification through testing and verification through monitoring. As already discussed the focus of this paper is on the latter type of verification. In the next section we will see how SXM models can be utilised for this purpose, for different kinds of Web services.

IV. RUN-TIME VERIFICATION

Run-time verification involves logging the traces of service execution and comparing the actual execution traces to the expected execution traces, i.e. monitoring. Monitoring can be performed on-line or off-line. In both cases the animator of the SXM model acts as an oracle for determining the expected execution traces. An execution trace for a Web service is a sequence of request-response (input-output) pairs. The JSXM animator acts as an oracle; it is provided with the request and it returns the expected response.

Fig. 2 depicts the different components and their interactions in the proposed run-time verification architecture. This architecture is implemented by a mediator intercepting the communication between the service client and the service provider. As discussed in the next section, this mechanism can be incorporated into the infrastructures of all different types of SOA stakeholders, i.e. service providers, service consumers, and service brokers.

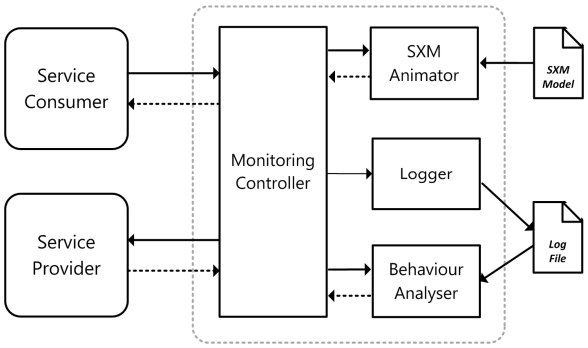

Figure 2. Run-time verification architecture.

When monitoring is carried out in on-line mode, the framework intercepts SOAP request messages and feeds them to the animator which runs an instance of the SXM model in parallel with the actual monitored service. Client requests, service responses and animator responses are saved in a log file. The behaviour analyser module performs the comparison of the service and animator responses and reports deviations. On-line monitoring can be useful in the case that the interested stakeholder wishes to be informed about discrepancies immediately and potentially take initiative to interrupt the interaction. In off-line monitoring mode the framework intercepts and logs client requests and service responses, while animation and validation are performed at a later point based on the logs. The following provides an extract of the XML log file which is created during run-time verification:

```
<call>
  <input name="openRequest" />
  <output name="openResponse" />
  <expected name="openResponse" />
</call>
<call>
  <input name="depositRequest">
    <arg name="amount" type="Int" value="1000" />
  </input>
  <output name="depositResponse">
    <result name="amount" type="Int" value="1000" />
  </output>
  <expected name="depositResponse">
    <result name="amount" type="Int" value="1000" />
  </expected>
  <input name="withdrawRequest">
    <arg name="amount" type="Int" value="2000" />
  </input>
  <output name="withdrawResponse">
    <result name="amount" type="Int" value="2000" />
  </output>
  <expected name="withdrawError"/>
</call>
```

The extract illustrates 3 requests: open, deposit, and withdraw and the corresponding actual and expected responses. Note that a violation occurs at the withdraw request; the animator produces a withdrawError since the withdraw input cannot be accepted (its guard condition balance > withdrawRequest.amount evaluates to false), whereas the actual (faulty) implementation has accepted the over-withdrawal.

The run-time verification framework spawns a different animator instance for each service instance. It is necessary that the service provider characterises beforehand the provided service using the classification provided in a previous section. The framework utilises this information in order to properly initialise the animator instance as follows.

For private-transient-state services the animator begins animating the SXM from its initial state and memory, since each service instance works on initialised state. In the private-transient state shopping cart example the shopping cart will be initially empty every time the service is invoked.

For private-persistent-state services (as the bank account example from the previous section) the animator needs to start animation from the last saved state and memory for the specific client or animate the previously saved log. This is illustrated in the sequence diagram of Fig. 3, where a new instance of the Animator is created and the saved log file is "replayed" in order to bring the animator to the last saved state. As already mentioned private-persistent state services require some kind of identification. This identification should also be used for loading the state of the animator or the log. After each transition the animator saves its current state and its current memory values associated with the identifier of the service client.

Shared-state services require a different treatment than private-state services. In these cases it is rather impractical if not impossible to model external factors that affect state variables. In the shopping cart with inventory support, one would need to model the whole inventory of existing and available items with their quantities. Even this would not suffice since other service clients will be able to change inventory quantities by purchasing items. So, complete modelling of shared-state services becomes intractable and unrealistic. In this case non-deterministic SXMs [10] are deployed. For operations operating on shared data all possible responses, memory updates and state transitions are modelled as different transitions which may occur non-deterministically; the same operation with identical input values can cause different behaviour (trigger different transitions which lead to different states or change the memory in a different way) and produce different outputs. Since the animator is not able to determine which transition should be triggered it needs to wait for the actual response from the service implementation. By matching the actual response with the outputs of one of the possible transitions (this is possible due to a property of SXM models called output-distinguishability) the JSXM animator can continue the animation of the SXM model by selecting the corresponding transition. If the response does not match any of the expected responses then non-conformance is reported. For example, the addItem(id, quantity) operation of a shopping cart, could either return the response "Item Added" or "Item Not Available" depending on whether the quantity for the item is available on stock or not. Both behaviours need to be modelled in a non-deterministic manner, as two possible transitions with the same precondition but possibly different post-states, outputs and memory updates. Depending on the response of the Web service, the animator will choose the transition to be followed in order to continue with the animation.

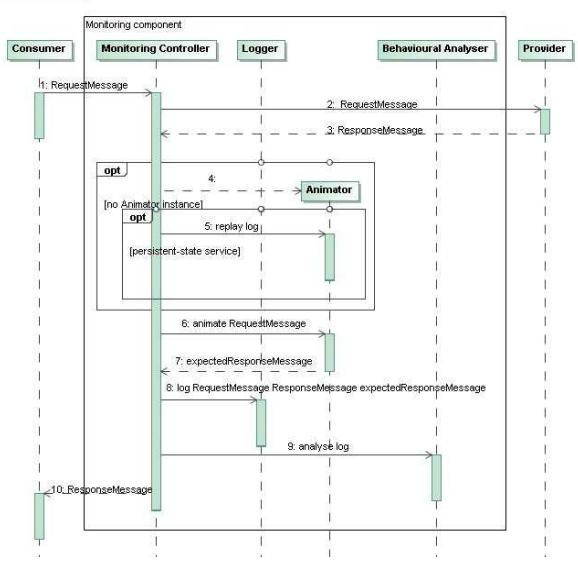

Figure 3. Sequence diagram of interactions between architecture components during run-time verification.

V. INTEGRATION OF RUN-TIME VERIFICATION INTO SOA INFRASTRUCTURES

Run-time verification of Web service behaviour can find applications in a wide range of contexts, and can be integrated within service-oriented infrastructures in a variety of configurations. One of the advantages of the run-time verification method that is introduced in this paper, in contrast to other approaches found in the literature, is that it can be utilised by all types of

stakeholders in a SOA environment, i.e. by service providers, service consumers, and service brokers. In the following subsections we examine some of the different reasons that motivate the incorporation of run-time verification in the infrastructures of different types of SOA stakeholders, and discuss some implementation alternatives for carrying out the integration.

A. Consumer-based monitoring and verification

In general, the incorporation of monitoring capabilities in the infrastructure of an entity that consumes Web services provided by some other entity allows the first to continuously verify if the latter meets certain requirements with regard to functional and non-functional aspects of service operation at all times. As already mentioned, in this paper we specifically focus on monitoring functional aspects of a service's operation (i.e. its behaviour) rather than non-functional ones (i.e. its response times, availability, etc). A typical situation where an entity would be interested in continuously monitoring and verifying the behaviour of a service it consumes is when that service needs to conform to a specific industry standard that prescribes a particular interaction protocol for the parties engaged in a business process. An example could be conformance to standardised or bespoke business processes such as those defined via ebXML Business Process Specification Schema or RosettaNet Partner Interface Process (e.g. defining how a stock replenishment business process should be realised).

There are several ways in which run-time monitoring and verification capabilities can be incorporated in the technical infrastructure of service consumers. The one that appears to be the most effective, since it requires no changes to the service provider's end, and trivial changes to the client application, is to create local wrapper Web services – one for each of the actual remote Web services that need to be monitored - and deploy them on a Web application server at the consumer's site. Instead of binding to the actual remote Web services the client application must be modified in order to bind to the wrapper services, which will in turn bind to the actual ones. The rationale is illustrated in Fig. 4.

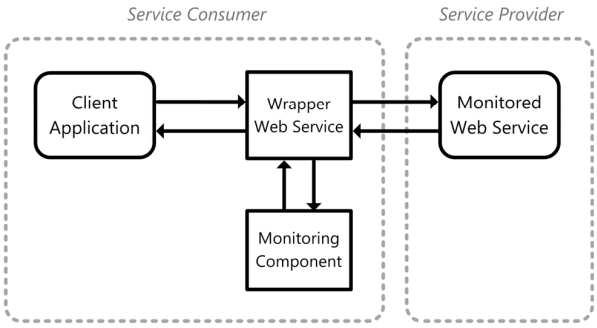

Figure 4. Integration of run-time verification at the consumer's site

During execution the wrapper service receives the incoming request messages, logs them, forwards them to both the real service being monitored and to the oracle (SXM animator), logs the obtained responses, and forwards the response of the monitored service to the consumer. The logs can be analysed during execution (on-line) or at a later stage (off-line), and appropriate actions can be taken automatically, such as terminating all interactions with a particular faulty service.

B. Provider-based monitoring and verification

Another possibility is to integrate our proposed approach for run-time monitoring and verification at the provider's infrastructure. The main motivation is being able to detect potential lapses in conformance with prescribed interaction protocols as early as possible, in order to take corrective actions that mitigate the effects of the situation before all consumers of a faulty service become affected. A conformance lapse of this kind may result from the release of a new version of a particular Web service in which defects were accidentally introduced but went undetected during regression testing, or from reasons beyond the control of the service provider, such as dependence on some faulty third party Web service. Continuous monitoring of the behaviour of all of their Web services enables providers to not only detect such errors, but also be able to present evidence of correct behaviour in case consumers raise misinformed claims when no errors actually exist.

Run-time monitoring and verification capabilities can be incorporated in the infrastructure of service providers by modifying and extending the functionality of the Web application servers that host the Web services (Tomcat, JBoss, etc). The rationale is illustrated in Fig. 5.

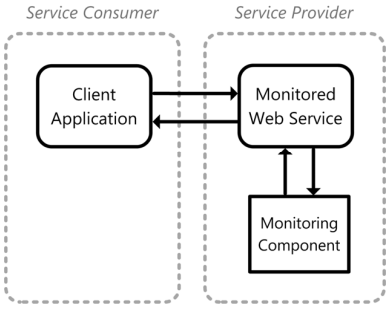

Figure 5. Integration of run-time verification at the provider's site

The application server intercepts incoming request messages, logs them, forwards them to the service being monitored and to the oracle (animator), and logs the obtained responses. Once again the logs can be analysed in an on-line or off-line mode. In case of detected deviation from expected behaviour the embedded monitoring component of the application server can alert the administrator so as to take corrective action and possibly also stop the service.

C. Broker-based monitoring and verification

An additional possibility is the incorporation of run-time verification in the infrastructure of a broker which acts as a trusted entity for both providers and consumers. Broker-based monitoring can be employed for preventing defective or non-compliant Web services from being discovered and reused. This is a particularly appealing scenario in the case of brokers which also act as certification authorities for published Web services, as in the approach proposed in [13, 14]. Run-time verification can complement publication-time verification and can

enable brokers to revoke the certification status from a non-compliant service before further damage is caused to prospective consumers.

Integration of run-time monitoring and verification at the broker's end can be achieved following the same approach as in consumer-based monitoring, i.e. through binding to intermediary wrapper Web services. The only difference is that the wrapper services are not to be hosted at the consumer's site, but at the broker's site (see Fig. 6).

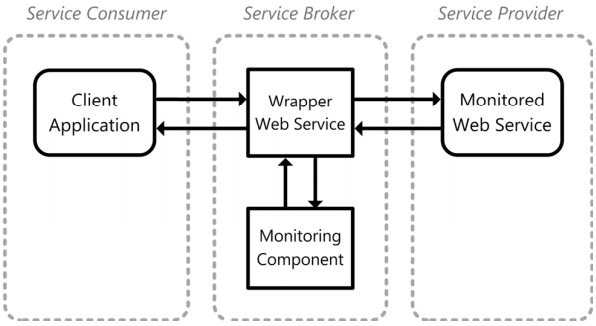

Figure 6. Integration of run-time verification at the broker's site

The service consumer is once again not meant to communicate with the actual services of the provider directly, but through the broker-based wrapper services which enable end-to-end monitoring. Nonetheless it should be noticed that such a scheme can potentially result in the creation of processing bottlenecks which would impair the scalability of the broker's infrastructure, so addressing performance-related aspects during the design of such a solution is crucial.

VI. Related Work

Different approaches have been proposed on run-time monitoring and behavioural conformance checking of conversational Web services. In general, these approaches are based on abstract specifications of expected service behaviour, which are used for conformance verification of the observed behaviour.

Mahbub and Spanoudakis [18] describe an approach for monitoring Web services in which properties to be monitored are expressed using event calculus. Behavioural properties of the services are automatically extracted from a BPEL process that orchestrates the services. The authors propose an architectural framework, consisting of a requirements editor, an events receiver, and a monitor. Conformance evaluation is performed offline (post-mortem), by exploiting techniques for integrity constraint checking in temporal deductive databases. This approach focuses on the behavioural properties of composition processes expressed in BPEL rather than on individual Web services.

Van der Aalst et al [1] and Rozinat et al [21] describe an approach to run-time monitoring and conformance evaluation based on Petri nets and message logs recorded during monitoring. Initially, the expected Web service behaviour is specified in an abstract BPEL process model. The BPEL specification is translated to a WF-net, which is a subclass of Petri nets. Also, the authors propose an approach to monitor and to correlate SOAP messages in order to construct events logs. Protocol conformance checking is performed by comparing the obtained event logs with the Petri net. Li et al [16] use pattern and scope operators from a language called Specification Pattern System (SPS) to express service interaction constraints regarding both the occurrence of individual events and sequences of events. These constraints are then represented as finite state automata (FSA) in order to facilitate analysis. A framework is described for validating the behaviour of the monitored Web service against the predefined constraints. In both previously mentioned approaches [1, 21, 16] Web service monitoring focuses on checking the fulfilment of control flow dependencies such as a missing response after a request. Input and output values or state variables are not considered. As a consequence, the operation invocation sequence open(), deposit(1000), withdraw(2000) is accepted as valid if only control flow dependencies are considered. In our approach, however, the same sequence is identified as invalid since the modelled bank account does not allow over-withdrawals; the state variable balance holds the actual account balance and the guard of the withdraw operation makes sure that the balance never becomes negative.

Lohman et al [17] propose run-time monitoring of Web service behaviour as part of a reliable service life-cycle, including reliable service registration and service discovery. Expected Web service behaviour is specified at the level of operations. They model data types visible at a service interface with class diagrams, while the behaviour of each operation is specified by graph transformation (GT) rules. Each GT rule describes the manipulation of object structures over the class diagram. Monitors are then generated automatically based on a model-driven approach, as described in Engels et al [9]; class diagrams are translated to Java code and GT rules are translated to JML assertions. The service operations are invoked by wrapper methods, which throw exceptions whenever pre-conditions or post-conditions are violated. The thrown exceptions are caught by the Web service client who then decides on the appropriate reaction. Preconditions and post-conditions which are expressed as conditions on operation inputs are adequate for expressing the functional behaviour of stateless services, but the behaviour of stateful services also depends on internal state variables. Thus their approach for monitoring stateful services is only applicable using server-side monitors which have access to Web service state. The extended finite state-based approach that we propose overcomes this obstacle by modelling the encapsulated state variables and thus allows the integration of the monitoring architecture both at consumer's site and at the provider's site.

Bianculli and Ghezzi [3] use algebraic specifications, to specify the expected service behaviour. The algebraic specifications consist of rewrite rules describing the behaviour of individual service operations. Interpreters such as CafeOBJ and Heureka are used for the symbolic execution of the specifications. Then, the outputs obtained from the Web service are compared with the evaluation results returned by the interpreter, in order to check for deviations. The authors also propose an architectural framework to enable run-time monitoring of individual conversational services orchestrated by a BPEL engine.

They weave aspects (using AspectJ) to the ActiveBPEL engine to extend it with a main interceptor, a specifications registry, and a monitor. The aspect oriented approach is utilised in order not to intervene in the business logic of the BPEL process. In this approach, as in our approach, apart from protocol checking the actual returned values from operation invocations are compared to the expected outputs. In contrast however with our state-based specifications, Web service state is modelled implicitly by using axioms which specify the equivalence of sequences of operations.

Furthermore, an important characteristic of our approach compared to all of the aforementioned approaches is its intuitiveness to the software engineer. We believe that a state-based description of service behaviour that is usually represented using a UML state-diagram or a UML Protocol State Machine can easily be transformed to a Stream X-machine model.

VII. Conclusions and Future Work

Service-oriented systems have an open architecture, which allows third-party services to be incorporated dynamically, raising the need for continuous verification. Testing Web services before deployment cannot guarantee that they will not deviate from their advertised behaviour during execution. Therefore, run-time verification of Web service behavioural conformance is considered critical. In this paper we proposed a novel run-time verification approach that utilises Stream X-machine models for specifying Web service behaviour. We presented a classification of Web services and investigated the application of run-time monitoring in different classes. In addition, we proposed an architecture that allows run-time verification of behavioural conformance for conversational Web services. Our presented approach has some significant advantages compared to other approaches for run-time monitoring and verification. Firstly, it allows for checking both the control flow of a Web service and the values of the data in the generated responses. Secondly, it can be utilised by all types of stakeholders in a SOA environment, i.e. by service providers, service consumers, and service brokers. Thirdly, it employs a state-based behavioural modelling formalism that is arguably more intuitive to the software engineer than other formalisms utilised by other approaches. As future work we intend to work on the validation of the approach with realistic conversational Web services falling into different categories.

References

[1] W.M.P.V.D. Aalst, M. Dumas, C. Ouyang, A. Rozinat, and E. Verbeek, "Conformance checking of service behavior," ACM Trans. Interet Technol., vol. 8, 2008, pp. 1-30.

[2] C. Atkinson, D. Stoll, H. Acker, P. Dadam, M. Lauer, and M. Reichert, "Separating per-client and pan-client views in service specification," Proceedings of the 2006 international workshop on Service-oriented software engineering, Shanghai, China: ACM, 2006, pp. 47-53.

[3] D. Bianculli and C. Ghezzi, "Monitoring conversational Web services," 2nd international workshop on Service oriented software engineering: in conjunction with the 6th ESEC/FSE joint meeting, Dubrovnik, Croatia: ACM, 2007, pp. 15-21.

[4] D. Brenner, C. Atkinson, O. Hummel, and D. Stoll, "Strategies for the Run-Time Testing of Third Party Web Services," Proceedings of the IEEE International Conference on Service-Oriented Computing and Applications, IEEE Computer Society, 2007, pp. 114-121.

[5] D. Dranidis, G. Eleftherakis and P. Kefalas, "Object-based Language for Generalized State Machines," Annals of Mathematics, Computing and Teleinformatics (AMCT), 1 (3), 2005, 8-17.

[6] D. Dranidis, D. Kourtesis, and E. Ramollari, "Formal verification of Web service behavioural conformance through testing," Annals Of Mathematics, Computing & Teleinformatics,, vol. 1, 2007, pp. 36-43.

[7] D. Dranidis, "JSXM: A Suite of Tools for Model-Based Automated Test Generation: User Manual." Technical Report WP-CS01-09, CITY College, 2009.

[8] S. Eilenberg, Automata, Languages and Machines, Academic Press, 1974.

[9] G. Engels, M. Lohmann, S. Sauer, and R. Heckel, "Model-Driven Monitoring: An Application of Graph Transformation for Design by Contract," Graph Transformations, 2006, pp. 336-350.

[10] R.M. Hierons and M. Harman, "Testing conformance of a deterministic implementation against a non-deterministic stream X-machine," Theor. Comput. Sci., vol. 323, 2004, pp. 191-233.

[11] M. Holcombe and F. Ipate, Correct systems: Building a business process solution, Springer-Verlag, 1998.

[12] E. Kapeti and P. Kefalas, "A Design Language and Tool for X-Machine Specification," Advances in Informatics, Proceedings of the 7th Hellenic Conference on Informatics (HCI '99), 2000.

[13] D. Kourtesis, E. Ramollari, D. Dranidis, and I. Paraskakis, "Discovery and Selection of Certified Web Services Through Registry-Based Testing and Verification," Pervasive Collaborative Networks, 2008, pp. 473-482.

[14] D. Kourtesis, E. Ramollari, D. Dranidis, and I. Paraskakis, "Increased Reliability in SOA Environments through Registry-Based Conformance Testing of Web Services," Special issue of Journal on Production Planning & Control on Engagement in Collaborative Networks, Taylor & Francis, 2009,(in press).

[15] G. Laycock, "The Theory and Practice of Specification-Based Software Testing," Thesis (PhD). Department of Computer Science, University of Sheffield, 1993.

[16] Z. Li, Y. Jin, and J. Han, "A Runtime Monitoring and Validation Framework for Web Service Interactions," Proceedings of the Australian Software Engineering Conference, IEEE Computer Society, 2006, pp. 70-79.

[17] M. Lohmann, L. Mariani, and R. Heckel, "A Model-Driven Approach to Discovery, Testing and Monitoring of Web Services," Test and Analysis of Web Services, 2007, pp. 173-204.

[18] K. Mahbub and G. Spanoudakis, "A framework for requirents monitoring of service based systems," Proceedings of the 2nd international conference on Service oriented computing, New York, NY, USA: ACM, 2004, pp. 84-93.

[19] E. Ramollari, D. Kourtesis, D. Dranidis, and A.J. Simons, "Towards reliable Web service discovery through behavioural verification and validation," Proceedings of the 3rd European Young Researchers Workshop on Service Oriented Computing, London: 2008.

[20] E. Ramollari, D. Kourtesis, D. Dranidis and A.J.H. Simons, "Leveraging Semantic Web Service Descriptions for Validation by Automated Functional Testing, The Semantic Web: Research and Applications, Springer LNCS 5554, 2009, 3-607.

[21] A. Rozinat and W.M.P.V.D. Aalst, "Conformance checking of processes based on monitoring real behavior," Inf. Syst., vol. 33, 2008, pp. 64-95.