Formal Verification of Web Service Behavioural Conformance through Testing

Dimitris Dranidis

Computer Science Department
CITY College
Affiliated Institution of the University of Sheffield
Tsimiski 13, 54624 Thessaloniki, Greece

dranidis@city.academic.gr

Dimitrios Kourtesis, Ervin Ramollari

SEERC - South East European Research Centre
17 Mitropoleos Str., 54624 Thessaloniki, Greece

{dkourtesis,erramollari}@seerc.org

Abstract—The value proposition of Web service technology lies in composability, reusability, and substitutability, a set of key characteristics that however give rise to major challenges when realising service-based systems, such as trustworthiness and interoperability. Thus, being able to verify that a Web service implementation conforms to certain functional or non-functional requirements is instrumental in engineering dependable service-based systems. This paper introduces a new approach to verifying the conformance of a Web service implementation against a behavioural specification, through the application of testing. We propose the use of Stream X-machines as an intuitive modelling formalism for constructing the behavioural specification of a stateful Web service, and propose the use of a method for deriving test cases from that specification in an automated way. The test generation method has been proven to produce complete sets of test cases that are guaranteed to reveal all non-conformance faults in a service implementation under test. The verification approach that we put forward can be applied in a range of different contexts and yield significant benefits for all types of stakeholders in a SOA environment, i.e. service providers, service consumers, and service brokers.

Index Terms—Service-Oriented Architecture, Web Service Testing, Test Case Generation, Formal Methods, Stream X-Machines.

I. INTRODUCTION

SERVICE-oriented computing is an emerging paradigm for distributed computing that is changing the way software applications are architected, realised, delivered, and consumed [11]. The term Service-Oriented Architecture (SOA) refers to a software architecture perspective where nodes on a network make computational resources available to other network nodes in the form of services. Services are self-contained, autonomous, highly reusable software components with programmatic interfaces that can be described, discovered and used independently of their underlying platform, implementation language, or software vendor.

The prevailing approach for realising SOA today is through Web services, primarily due to the way in which Web services naturally implement the SOA philosophy of loose coupling and reusability and promote interoperability by leveraging on widely accepted standards like WSDL, SOAP, and UDDI.

As with all types of software artefacts, testing is an integral component of the Web services development lifecycle. In addition, a number of distinctive characteristics that Web services have, such as their value proposition for reusability, composability, and substitutability, but also key challenges like trustworthiness and interoperability, make testing indispensable for post-development lifecycle phases as well. Notably, this applies to services that either stand alone as individual components, or are orchestrated within a Web service composition.

Testing is a means to verify whether a Web service conforms to certain functional or non-functional requirements. Non-functional requirements may reflect constraints on performance, security, or other Quality of Service (QoS) properties. Functional requirements relate to the behaviour that a Web service is expected to exhibit when consumed. Different types of testing techniques can be applied depending on the requirements and the type of conformance to be asserted, but in most of the cases, testing can only be performed in

a black-box manner, since a Web service's source code is usually unavailable. Verifying the conformance of a Web service's behaviour against consumer requirements is especially challenging in the case of stateful Web services in which an interaction protocol is assumed. As will be presented in the next section of this paper, a number of research works have attempted to address this type of problem with the help of formal methods. A common characteristic in all of these works is the construction of a model explicating the behaviour of a service, and the generation of test cases to assert that a service implementation indeed meets the specification requirements.

This paper introduces a new approach to verifying the conformance of a Web service implementation against a behavioural specification. We propose the use of Stream X-machines [10] as a modelling formalism for constructing the behavioural specification of a service and generating test cases to verify that one or more service implementations conform to that specification. Stream X-machines are a subclass of X-machines [5] that extend Finite State Machines (FSMs) and are supported by a test generation method that is guaranteed to reveal all faults in an implementation under test, given that certain realistic conditions hold. The Web service test set is represented as a sequence of operation invocations with associated inputs and expected outputs. By applying the generated set of tests to a stateful Web service implementation and evaluating its responses one can conclude if it is behaviourally-equivalent to the original specification. As will be discussed in a subsequent section of this paper, the application of this verification approach can yield significant benefits for all stakeholders in a service-based environment, i.e. service providers, service consumers, and service brokers.

The rest of the paper is structured as follows. Section 2 presents a summary of related work on this topic, i.e. on approaches for modelling the behaviour of stateful Web services and for automatically or semi-automatically generating tests to assert some form of behavioural conformance. Section 3 provides an overview of the verification approach put forward in this paper, stating the basic assumptions on which it is based. Section 4 introduces the Stream X-machine modelling formalism and demonstrates the approach for constructing a behavioural specification and the procedure of generating test-sets, through a stateful Web service example scenario. Section 5 discusses the benefits of the verification approach in different settings. Section 6 concludes this paper by summarising the most important aspects of the presented work, discussing the strengths and limitations of the proposed approach, and suggesting directions for future work.

II. RELATED WORK

Currently, most formal approaches to Web services verification focus on how to guarantee desired properties in Web service compositions such as correctness, fault-tolerance, deadlock avoidance, reachability, liveness, etc. [15], [7], [22], [21], [23]. Instead of verifying the individual Web services participating in a composition, most of these approaches assume that they are correct, and instead focus on verifying the composition as a whole. Notably, the aforementioned approach does not necessarily guarantee the trustworthiness of the resulting composition. In contrast, the work presented in this paper addresses formal verification and test case generation at the level of individual Web services (component services). The rest of this section discusses some related state-of-the-art work that shares the same focus.

Tsai et al [26] propose extending WSDL descriptions with additional information that aids testing, including input-output dependency, invocation sequences, hierarchical functional description, and sequence specifications. However, no method is given for generating test cases from this augmented information. In subsequent work [25] Tsai et al describe a specification-based validation and verification technique, where the specification is written in OWL-S, and the method of Boolean expression analysis is used to extract the full scenario coverage of Boolean expressions. The results are then provided as input to a tool named Swiss Cheese in order to generate both positive and negative test cases. The test cases can be used for verifying the correctness of the output produced by atomic service operations but cannot be applied for evaluating complex behaviour in stateful services which results from a sequence of operation invocations. As a result, the testing procedure lacks coverage for behavioural aspects, and cannot be considered sufficient for verifying behavioural conformance of stateful Web services.

Heckel and Mariani [9] employ graph transformation rules in a service discovery and matchmaking approach, for modelling both the behaviour of the provided service and the consumer's requirements. They propose a test case derivation method based on the service model that the provider supplies with the service description, in order to verify that the actual service implementation is compliant with the attached model. The resulting test cases include both single operation invocations as well as sequences of operations. Nevertheless, test case derivation is based on partition testing, which is a domain-dependent strategy that relies on the tester's experience to split the input domain into subsets. This may lead to non-uniform and biased tests. Moreover, the test case derivation algorithm considers only data flow for the generation of the test cases and ignores

control flow criteria, which are an important factor when verifying behavioural compliance.

Keum et al [19] present an approach to modelling and testing stateful Web services based on Extended Finite State Machines (EFSMs), a Finite State Machines variant that has been extended with memory, as well as with predicate conditions and computing blocks for state transitions. A procedure is described for semi-automatically deriving the EFSM model from a WSDL specification and additional user input. The model covers behavioural aspects of stateful Web services, and the resulting test cases represent sequences of invocations of Web service operations. Compared to other approaches in the literature, the test case generation algorithm that is employed provides better testing coverage, since it considers both control flow and data flow [1]. The authors provide experimental results showing that their method has the potential to find more faults compared to other methods, but notably, with a resulting test case set that is much larger and takes more time to execute.

Of all the aforementioned approaches, the one by Keum et al [19] seems to be the most mature solution to the problem of verifying the behavioural conformance of stateful Web services, and the one that is most relevant to our work. Nevertheless, in contrast to the test case generation method that is employed in [19], the method that our approach relies on can generate a complete test-set that is proven to reveal all faults in an implementation under test [13], given certain realistic assumptions [10].

III. DESCRIPTION OF THE APPROACH

The approach that we put forward in this paper, as illustrated in Figure 1, comprises of five phases: (i) constructing a model of a Web service's intended behaviour, (ii) generating test cases based on this model, (iii) mapping the abstract input/output messages in the test cases to concrete messages in a Web service implementation under test, (iv) feeding the test cases to the respective Web service operations via a SOAP interface, and lastly, (v) evaluating the service responses with respect to the expected output. If the testing process does not reveal any errors, the Web service implementation can be considered behaviourally-equivalent to the Web service specification.

The intended behaviour of a Web service is formally specified using the Stream X-machine formalism [10]. Stream X-machines provide the modelling constructs necessary for precisely defining the control flow and temporal ordering of operations in a stateful Web service; responses of operations in a stateful Web service depend not only on the input provided by the consumer, but also on the internal state of the service, itself

a result of previous operation invocations. The next section provides an example of a stateful Web service (the `ShoppingCart` Web service) whose desired behaviour is modelled using a Stream X-machine.

A Stream X-machine based testing method [13], [10] that extends the W-method [2], enables us to derive a complete finite set of test cases that is proven to find all faults in the implementation. The method requires that the models satisfy certain design for test conditions, i.e. they are complete with respect to memory and output distinguishable. The outcome of the test generation algorithm is a finite set of input and expected output sequences.

The inputs and expected outputs need to be mapped to concrete executable test cases that can be processed by a testing engine, in order to interact with the Web service under test and provide the results. A number of commercial and non-commercial Web service testing tools are available for this purpose, such as SOAtest by Parasoft [24], and Coyote, described in [27].

The testing method guarantees that, if the input sequences fed to the Web Service implementation under test produce the expected results, then the implementation conforms to the specification.

For our approach to be applicable, we assume that the operations of the Web service under test follow the request-response pattern, i.e. they accept a request (input) message from the invoker and return a response (output) message. This makes it possible to fulfil the output distinguishable design for test condition, i.e. any two different processing functions should produce different outputs on each memory/input pair. As a result it is possible to tell from the outputs which processing functions have been activated during an execution path. We don't consider this assumption to be too restrictive, since request-response patterns are the normal case for Web service operations.

IV. MODELLING WEB SERVICES AS STREAM X-MACHINES

A. Stream X-Machines

Stream X-machines (SXMs) [20], [10] is a computational model capable of modelling both the data and the control of a system. SXMs are special instances of the X-machines introduced by Eilenberg [5]. They employ a diagrammatic approach of modelling the control by extending the expressive power of finite state machines. In contrast to finite state machines, SXMs are capable for modelling non-trivial data structures by employing a memory, which is attached to the state machine. Additionally, transitions between states are not labelled with simple input symbols but with processing functions. Processing functions represent internal system
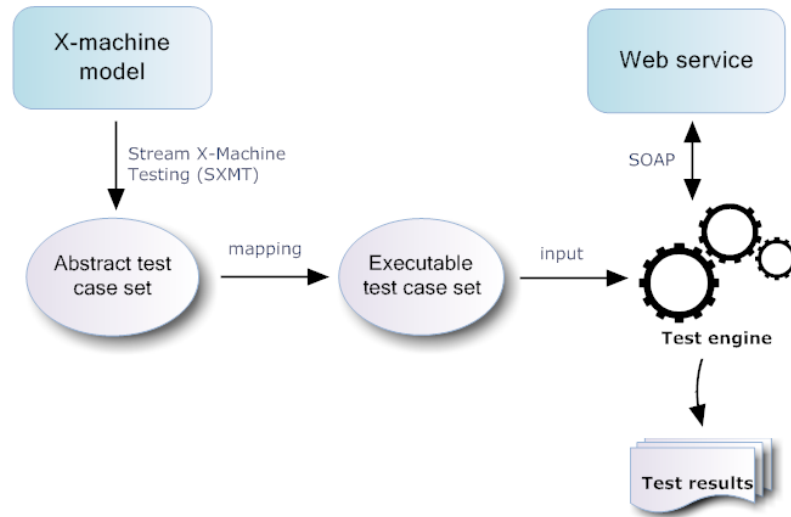
Fig. 1.   Behavioural conformance verification of Web services through testing.

transitions triggered by input symbols under specific memory conditions, and produce output symbols while modifying the memory.

The benefit of the addition of a memory structure is that state explosion is avoided and the number of states is reduced to those states which are considered critical for the correct modelling of the system.

A (deterministic) SXM is defined as follows [10]: $SXM = (\Sigma, \Gamma, Q, M, \Phi, F, q_0, m_0)$ 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 SXM, 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, $\phi : \Sigma \times M \to \Gamma \times M$;
- F is the next state partial function that given a state and a function from the type Φ, provides the next state, $F : Q \times \Phi \to Q$ (F is often described as a transition state diagram);
- q_0 and m_0 are the initial state and memory respectively.

The sequence of transitions (path) caused by the stream of input symbols is called a computation. The computation halts when all input symbols are consumed. The result of a computation is the sequence of outputs produced by this path.

The SXM models can be thought to apply in similar cases where Statecharts [8] and other similar models do. However, apart from being formal as well as proven to possess the computational power of Turing machines [10], SXMs have the significant advantage of offering a testing method [10], [13] that ensures conformance of an implementation to a specification. This method generates test sets for a system specified as a SXM whose application ensures that the system behaviour is identical to that of the specification, provided that the system is made of fault-free components and some explicit design for test requirements are met.

In order to assist the application of Stream X-machines the XMDL (X-Machine Definition Language) language was introduced in [14] and fully developed in Kefalas [16]. XMDL serves as a common language for the development of tools supporting Stream X-machines [17]. An extension of XMDL to support an object-based notation was suggested in [3]. The object-based extension, called XMDL-O, enables an easier and more readable specification of Stream X-machines and is employed in this paper for the specification of the example Web service.

B. The Shopping Cart Service Example and its SXM Model

1) The Web Service Example: The example we use to illustrate our approach is a simplified version of a Web service that is intended to provide the backend functionality of a shopping cart to client applications. The service allows a client to perform authentication, add items to a shopping cart or remove them, and proceed to checkout. Similar Web services providing functionality for storing and retrieving e-commerce orders are already in use and can be found on-line.[1] [2]

---

[1]http://webservices.amazon.com/AWSECommerceService/
[2]http://www.xwebservices.com/Web_Services/XWebCheckOut/

The `ShoppingCart` Web service provides the following operations:

- The `login` operation allows authentication for using the service. It is invoked with the input message `LoginRequest` consisting of the username and the password of the user. The request message is represented as:

  $$LoginRequest(user, pwd)$$

  The operation sends back the response message `LoginResponse(result)`, where result is a boolean value; true indicates successful authentication.

- The `addToCart` operation adds an item to the shopping cart. It is invoked with the input message `AddToCartRequest` consisting of the identifier of the item to be added. The request message is represented as:

  $$AddToCartRequest(itemId)$$

  The operation sends back the response message `AddToCartResponse(itemId)`. It is assumed that all item identifiers are valid and correspond to products that may be purchased.

- The `clearCart` operation removes all items from the shopping cart. It is invoked with the simple request message `ClearCartRequest` represented as

  $$ClearCartRequest()$$

  and it sends back the response message `ClearCartResponse()`.

- The operation `checkout` completes the shopping process. It is invoked with the simple request message `CheckoutRequest` represented as:

  $$CheckoutRequest()$$

  and it sends back the response message `CheckoutResponse()`.

2) Stateful Behaviour: The `ShoppingCart` service is a stateful Web service; the availability of its operations depends on the internal state of the service. For instance, the client is not allowed to perform any operation before authenticating, and checking out only makes sense with a non-empty cart. These two aspects of the stateful behaviour of the service are depicted in the state transition diagram of the SXM specification (Figure 2).

It has to be noted that the transitions on the diagram do not correspond to operations or messages of the Web
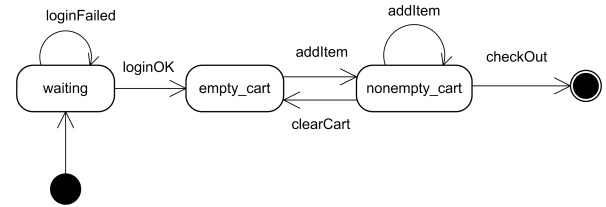

Fig. 2. State transition diagram for the `ShoppingCart` Web service.

service but to processing functions as specified in a subsequent section. Furthermore, some transitions that represent exceptional behaviour are not shown in the diagram for the sake of clarity. For instance, attempting to invoke the operation `addItem` while the service is found at state `waiting`, will exercise the self-transition `faultyAddItem`. Similar transitions exist for the rest of the operations and the states.

3) The Memory: The memory in the `ShoppingCart` service example is used to store information about valid user accounts and the contents of the shopping cart. The XMDL-O code in Figure 3 shows the definition of accounts as a set of Account objects and the cart as a set of item identifiers (strings). For the purpose of testing the system we assume that there are two valid user accounts.

```
#class Account {
    username: string,
    password: string,
}.

#objects:
    account1: Account,
    account2: Account,
    accounts: set_of Account,
    cart: set_of string.

#init_values:
    account1.username <- "usr1",
    account1.password <- "pwd1",
    account2.username <- "usr2",
    account2.password <- "pwd2",
    accounts <- {account1, account2},
    cart <- emptySet.
```

Fig. 3. XMDL-O code for the specification of the memory.

4) Specification of the Processing Functions: State transitions in SXMs are labelled with processing functions. A processing function is triggered by an input event when a specified guard condition holds, produces some output, and potentially updates (modifies) the memory. The updating of the memory consists of a sequence of assignments as specified in the `update` part of the processing function definition.

```
#fun loginOK( LoginRequest(?usr, ?pwd) ) =
    if ?account \= null and ?pwd = ?account.password
    then ( LoginResponse(true) )
    where
        ?account <- head (select(username = ?usr, accounts)).

#fun loginFailed( LoginRequest(?usr, ?pwd) ) =
    if ?account = null or ?pwd \= ?account.password
    then ( LoginResponse(false) )
    where
        ?account <- head (select(username = ?usr, accounts)).

#fun addItem( AddToCartRequest(itemId) ) =
    then ( AddToCartResponse() )
    update
        cart <- itemId addsetelement cart.

#fun clear( ClearCartRequest() ) =
    then ( ClearCartResponse() )
    update
        cart <- emptySet.

#fun checkOut( CheckOutRequest() ) =
    if cart \= emptySet
    then ( CheckOutResponse() ).
```

Fig. 4. XMDL-O code for the definition of processing functions.

The XMDL-O code in Figure 4 shows the definition of processing functions. When modelling Web services, the inputs and the outputs of the processing functions correspond intuitively to request and response messages of Web services respectively.

C. Test Generation

The greatest benefit of modelling systems with SXMs it the existence of a test generation method which under certain assumptions [12], [10], it is proven to find all faults in the implementation. The testing method is a generalization of the W-method [2]. It works on the assumption that the system specification and the implementation can be both represented as Stream X-machines with the same type Φ (i.e. both specification and implementation have the same processing functions) and Φ satisfies the following design for test conditions: completeness with respect to memory (all processing functions can be exercised from any memory value using appropriate inputs) and output distinguishability (any two different processing functions will produce different outputs if applied on the same memory/input pair).

When the above requirements are met, the Stream X-machine testing method may be employed to produce a complete test set of input sequences which can be used for the verification of the implementation. In fact it is proved that only if the specification and the implementation are behaviourally equivalent, the test set produces identical results when applied to both of them. Otherwise it is guaranteed that it will reveal the faults in the implementation.

The first step to constructing the test set of input sequences is based on applying the W-method [2] on the associated finite state automaton of the SXM, by considering processing functions as simple inputs. The test set X for the associated automaton consists of sequences of processing functions and it is given by the formula:

$$X = S(\Phi^{k+1} \cup \Phi^k \cup \ldots \cup \Phi \cup \{\epsilon\})W$$

where W is a characterization set, S a state cover of the associated finite state automaton, and k is the estimated difference of states between the implementation and the specification. A characterization set is a set of sequences of processing functions for which any two distinct states of the machine are distinguishable and a state cover is a set of sequences of processing functions such that all states are reachable from the initial state. The sets W, S, and X (for $k = 0$) for the ShoppingCart Web service example are presented in Figure 5.

The sequences of processing functions from the test-set X have to be converted to sequences of inputs. This is achieved by the fundamental test function as described in [10]. For instance, the sequence of processing functions $\langle \text{loginOK}, \text{addItem}, \text{addItem} \rangle$ is

$$W = \{\langle \mathtt{loginOK} \rangle, \langle \mathtt{addItem} \rangle, \langle \mathtt{checkOut} \rangle\}$$

$$S = \{\langle \rangle, \langle \mathtt{loginOK} \rangle, \langle \mathtt{loginOK, addItem} \rangle, \langle \mathtt{loginOK, addItem, checkOut} \rangle\}$$

$$
\begin{aligned}
X \quad = \{ \quad &\langle \mathtt{loginOK} \rangle, \langle \mathtt{addItem} \rangle, \langle \mathtt{checkOut} \rangle, \langle \mathtt{loginOK, loginOK} \rangle, \\
&\langle \mathtt{loginFailed, loginOK} \rangle, \langle \mathtt{addItem, loginOK} \rangle, \langle \mathtt{clearCart, loginOK} \rangle, \\
&\langle \mathtt{checkOut, loginOK} \rangle, \langle \mathtt{loginOK, addItem} \rangle, \langle \mathtt{loginOK, checkOut} \rangle, \\
&\langle \mathtt{loginOK, loginOK, loginOK} \rangle, \langle \mathtt{loginOK, loginFailed, loginOK} \rangle, \\
&\langle \mathtt{loginOK, addItem, loginOK} \rangle, \langle \mathtt{loginOK, clearCart, loginOK} \rangle, \\
&\langle \mathtt{loginOK, checkOut, loginOK} \rangle, \langle \mathtt{loginOK, loginOK, addItem} \rangle, \\
&\langle \mathtt{loginOK, loginFailed, addItem} \rangle, \langle \mathtt{loginOK, addItem, addItem} \rangle, \\
&\langle \mathtt{loginOK, clearCart, addItem} \rangle, \langle \mathtt{loginOK, checkOut, addItem} \rangle, \\
&\langle \mathtt{loginOK, loginOK, checkOut} \rangle, \langle \mathtt{loginOK, loginFailed, checkOut} \rangle, \\
&\langle \mathtt{loginOK, addItem, checkOut} \rangle, \langle \mathtt{loginOK, clearCart, checkOut} \rangle, \\
&\langle \mathtt{loginOK, checkOut, checkOut} \rangle, \ldots \}
\end{aligned}
$$

Fig. 5.   The sets W, S, and X (for $k = 0$). Note that X is not completely presented.

converted to the following sequence of inputs (Web service messages with specific data values):

$$
\begin{aligned}
\langle &\mathtt{LoginRequest("usr1","pwd1")}, \\
&\mathtt{AddToCartRequest("912")}, \\
&\mathtt{AddToCartRequest("875")} \rangle
\end{aligned}
$$

To complete the test generation process and enable a testing engine to execute the test cases, these abstract test cases have to be mapped to the respective operations and messages of the Web service implementation under test.

V. DISCUSSION

Verifying the conformance of a Web Service's behaviour against a formal specification has a number of important applications in the Web services realm, and can yield benefits for all different types of stakeholders in a SOA environment.

For service consumers who are considering one or more candidate services to choose from, verification can help determine if some service fits the business process of the consumer before it gets actually used (try-before-you-buy scenario), or before it replaces one of the services already participating in a service composition (try-before-you-replace scenario) [6].

For service providers who are outsourcing the development of a Web service implementation to a third-party, or are making frequent version releases of a Web service to introduce features or correct errors,

verification can help detect inconsistencies with respect to the original specifications or some earlier version of the service (try-before-you-provide scenario).

For service brokers who perform matchmaking among service requests and service advertisements, verification can help determine if a service advertisement has been ill-specified due to a mistake on behalf of the provider, or even malicious intent. This would allow the broker to decide if a service advertisement should be admitted for publication (try-before-you-publish scenario) [9], or returned following the submission of a request (try-before-you-suggest scenario).

VI. CONCLUSIONS

This paper introduces a new approach for verifying the conformance of a Web service implementation against a formal behavioural specification. We propose the use of Stream X-machines [10] as an expressive and intuitive formalism that is suitable for modelling the behaviour of complex Web services adhering to a stateful transaction protocol.

A significant advantage of Stream X-machines over Finite State Machines (FSMs) is their capacity to model truly complex systems without the state explosion occurring in FSMs. However, the real strength of this formalism is in testing. The Stream X-machine-based test generation method that our approach is building on can be proven to reveal all errors in an implementation under test [13], given that certain realistic assumptions hold [10]. Furthermore, this software verification approach has been already applied in a variety of contexts

in the past [18], [4] and can be readily supported by a number of tools [17].

Future work will focus on relaxing some of the restrictive assumptions in our approach, such as the lack of support for message exchange patterns other than request-response, or the requirement for structural equivalence among the interface of the Web service implementation under test and the specification model. Moreover, we will seek means to automate the process of mapping abstract to concrete input/output messages by extending the approach to include support for Semantic Web Services with SAWSDL interfaces and semantic annotations on input/output messages based on Description Logics (DL).

REFERENCES

[1] C. Bourhfir, R. Dssouli, E. Aboulhamid, and N. Rico. Automatic executable test case generation for extended finite state machine protocols. In *Proceedings of IWTCS'97*, pages 75–90, 1997.

[2] T. S. Chow. Testing software design modelled by finite state machines. *IEEE Transactions on Software Engineering*, 4:178–187, 1978.

[3] D. Dranidis, G. Eleftherakis, and P. Kefalas. Object-based language for generalized state machines. *Annals of Mathematics, Computing and Teleinformatics (AMCT)*, 1(3):8–17, 2005.

[4] D. Dranidis and K. Tigka. Supporting test case generation for extreme programming: a novel approach. In *Proceedings of the 10th Panhellenic Conference on Informatics (PCI 2005), Greece*, November 2005.

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

[6] M. D. Ernst, J. H. Perkins, and R. Lencevicius. Detection of web service substitutability and composability. In *Proceedings of the 1st International Workshop on Web Services Modeling and Testing (WS-MaTe 2006)*, June 2006. Palermo, Italy.

[7] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based verification of web service compositions. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE'03)*, 2003.

[8] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[9] R. Heckel and L. Mariani. Automatic conformance testing of web services. In *FASE 2005*, pages 34–48. Springer, 2005.

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

[11] M. N. Huhns and M. P. Singh. Service-oriented computing: Key concepts and principles. *IEEE Internet Computing*, 9(1):75–81, January 2005.

[12] F. Ipate. *Theory of X-machines with Applications in Specification and Testing*. PhD thesis, Department of Computer Science, University of Sheffield, 1995.

[13] F. Ipate and M. Holcombe. An integration testing method that is proven to find all faults. *International Journal of Computer Mathematics*, 63:159–178, 1997.

[14] E. Kapeti and P. Kefalas. A design language and tool for X-machine specification. In *Proceedings of the 7th Panhellenic Conference on Information Techology, Greek Computer Society, Ioannina*, 1999.

[15] R. Kazhamiakin and M. Pistore. A parametric communication model for the verification of BPEL4WS compositions. In *Proceedings of WS-FM'05*, 2005.

[16] P. Kefalas. X-machine description language: User manual, version 1.6. Technical Report WP-CS07-00, CITY College, 2000.

[17] P. Kefalas, G. Eleftherakis, and A. Sotiriadou. Developing tools for formal methods. In *Proceedings of the 9th Panhellenic Conference in Informatics*, pages 625–639, November 2003.

[18] P. Kefalas, M. Holcombe, G. Eleftherakis, and M. Gheorghe. A formal method for the development of agent-based systems. In V. Plekhanova, editor, *Intelligent Agent Software Engineering*, pages 258–273. Springer, 2005.

[19] C. Keum, S. Kang, and I. Y. Ko. Generating test cases for web services using extended finite state machine. In *TestCom 2006*, pages 103–117. Springer, 2006.

[20] G. Laycock. *The Theory and Practice of Specification-Based Software Testing*. PhD thesis, Dept of Computer Science, Sheffield University, UK, 1993.

[21] D. A. Menasce. Composing web services: A QoS view. *IEEE Internet Computing*, November-December 2004.

[22] S. Nakajima. Model-checking verification for reliable web service. In *Proceedings of the First International Symposium on Cyber Worlds (CW'02)*, pages 378–385, November 2002.

[23] S. Narayanan and S. McIlraith. Simulation, verification and automated composition of web services. In *Proceedings of the International World Wide Web Conference (WWW2002)*, pages 77–88, 2002.

[24] Parasoft. *SOATest Data Sheet*. www.parasoft.com.

[25] W. T. Tsai, Y. Chen, and R. Paul. Specification-based verification and validation of web services and service-oriented operating systems. In *Proceedings of 10th IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS'05)*, pages 139–147, February 2005.

[26] W. T. Tsai, R. Paul, Y. Wang, C. Fan, and D. Wang. Extending WSDL to facilitate web services testing. In *Proceedings of the 7th IEEE International Symposium on High Assurance Systems Engineering (HASE'02)*, 2002.

[27] W. T. Tsai, R. Paul, S. Weiwei, and C. Zhibin. Coyote: an XML-based framework for web services testing. In *Proceedings of the 7th IEEE International Symposium on High Assurance Systems Engineering (HASE'02)*, 2002.