*promoting access to White Rose research papers*

White Rose Research Online

**Universities of Leeds, Sheffield and York**
**http://eprints.whiterose.ac.uk/**

This is an author produced version of a paper published in **Lecture Notes in Computer Science.**

White Rose Research Online URL for this paper:
http://eprints.whiterose.ac.uk/10537/

**Published paper**

Ramollari, Ervin, Kourtesis, Dimitrios, Dranidis, Dimitris and Simons, Anthony J.H. (2009) *Leveraging Semantic Web Service Descriptions for Validation by Automated Functional Testing.* In: The Semantic Web: Research and Applications. Lecture Notes in Computer Science, 5554/2009 . Springer , Berlin / Heidelberg, pp. 593-607.

http://dx.doi.org/10.1007/978-3-642-02121-3_44

*White Rose Research Online*
*eprints@whiterose.ac.uk*

# Leveraging Semantic Web Service Descriptions for Validation by Automated Functional Testing

Ervin Ramollari[1], Dimitrios Kourtesis[1], Dimitris Dranidis[2], Anthony J.H. Simons[3]

[1] South East European Research Centre (SEERC),
Research Centre of the University of Sheffield and CITY College
Mitropoleos 17, 54624, Thessaloniki, Greece
erramollari@seerc.org, dkourtesis@seerc.org

[2] Computer Science Department, CITY College,
Affiliated Institution of the University of Sheffield
Tsimiski 13, 54624 Thessaloniki, Greece
dranidis@city.academic.gr

[3] Department of Computer Science, University of Sheffield
Regent Court, 211 Portobello Street, Sheffield S1 4DP, UK
a.simons@dcs.shef.ac.uk

**Abstract.** Recent years have seen the utilisation of Semantic Web Service descriptions for automating a wide range of service-related activities, with a primary focus on service discovery, composition, execution and mediation. An important area which so far has received less attention is service validation, whereby advertised services are proven to conform to required behavioural specifications. This paper proposes a method for validation of service-oriented systems through automated functional testing. The method leverages ontology-based and rule-based descriptions of service inputs, outputs, preconditions and effects (IOPE) for constructing a stateful EFSM specification. The specification is subsequently utilised for functional testing and validation using the proven Stream X-machine (SXM) testing methodology. Complete functional test sets are generated automatically at an abstract level and are then applied to concrete Web services, using test drivers created from the Web service descriptions. The testing method comes with completeness guarantees and provides a strong method for validating the behaviour of Web services.

**Keywords:** Semantic Web Services, Web service testing, Service Validation

## 1 Introduction

The vision in Semantic Web Services (SWS) research [13], [14] is to bring formal semantics into the Web services realm such that service characteristics can be explicated in an unambiguous, computer-interpretable manner that allows for automating a broad range of service design-time and run-time activities. Recent years have seen the development of numerous SWS frameworks that aim to address this goal. The most prominent ones have been OWL-S [15], WSMO [4], and WSDL-S [1]. The latter served as the basis for the recently ratified W3C Recommendation of SAWSDL [7] which is currently the only standard in the area. Research around SWS frameworks has mostly focused on the development of methods and tools for enabling automated Web service discovery, composition, and execution [13], [14], while the areas of service testing and validation have remained largely unexplored.

The ability to validate that a Web service implementation conforms to given functional/behavioural specifications through functional testing is instrumental in engineering dependable service-based systems. However, the process of Web service testing is currently highly error-prone and laborious, as most of the activities, and especially the construction of test cases, need to be carried out manually. In this paper we show that in the presence of service descriptions with formal semantics, the process of generating test cases and performing validation through testing can be automated to a great extent. In particular, we propose a method for deriving a formal specification of a Web service's behaviour as a Stream X-machine (SXM) [9] model, by transforming ontology-based and rule-based descriptions of the service's inputs, outputs, preconditions, and effects (IOPE). The derived SXM model is utilised for automatically generating abstract test cases, which are executed against the deployed Web service to perform validation.

An important aspect of our approach is the strength of the validation method, which stems from the completeness guarantees that the SXM testing method can provide. The SXM testing method is guaranteed to generate a complete, finite set of test cases able to reveal all inconsistencies among an SXM specification and an implementation under test. Another important aspect is the adoption and promotion of standards. The semantic descriptions of Web service IOPE from which the SXM model is derived, are encoded using an existing and a forthcoming standard from the Semantic Web technology stack. OWL [16] is employed for specifying the service’s data model, and RIF-PRD (Rule Interchange Format – Production Rule Dialect) [21] is employed for specifying the service’s functionality. As a forthcoming standard for rule interchange, RIF [22] can assist in overcoming the impedance mismatch among today’s heterogeneous SWS frameworks, and can enable our approach to remain generic and even reusable with different frameworks like WSMO or OWL-S.

The rest of this paper is organised as follows. Section 2 presents an overview of related works in the area of Web service testing, validation and verification that involves the use of semantic service descriptions. Section 3 discusses how stateful Web service behaviour can be represented in terms of IOPE, and how Semantic Web technologies such as OWL and RIF can be employed for this purpose, with the help of an example Web service. Section 4 provides an overview of the Stream X-machine modelling formalism which we adopt for modelling Web service behaviour and generating test cases in an automated way. Section 5 provides a detailed step-by-step description of the method of deriving an SXM model from the IOPE-based service description. Section 6 discusses the overall testing approach, the generation of test cases from the SXM model, and the way in which tests can be executed against a Web service. Finally, section 7 concludes the paper with a summary of the key points and an outlook to future research objectives.

## 2 Related Work

Methods and tools for Web service testing, verification and validation are essential for engineering and managing dependable service-based software applications, and are receiving increasing attention. In this section we consider related research works that address the validation of Web services through model-based testing, where a model is used to generate a set of test cases.

Heckel and Mariani [8] employ graph transformation rules to model the behaviour of individual operations of stateful Web services in terms of preconditions and updates on the world state. The graph transformation rules are used to generate test cases using domain partitioning and data-flow criteria. The method is used in a registry-based testing approach to validate conformance of advertised services.

Bertolino et al [2] also propose a registry-based testing approach where the provider augments the WSDL document with behavioural descriptions in a UML 2.0 Protocol State Machine (PSM) diagram, which is translated to a Symbolic Transition System (STS). The registry uses the STS model to generate a set of test cases, which are run on the Web service under test for validating behavioural conformance.

Wang et al [23] use the OWL-S description of a composite service process as a starting point to derive an equivalent Petri-net model through an informal transformation algorithm. The Petri-net model is then used to generate test cases and perform model-checking.

Narayanan and McIlraith [17] also describe a manual method to construct a Petri net from descriptions in DAML-S (the precursor of OWL-S). They propose using this model for a number of activities, including testing. However, no testing approach and test-case generation algorithm is described, since the authors assume that a test suite already exists and the model serves as a test oracle to predict the expected outputs.

Sinha and Paradkar [19] utilise IOPE-based descriptions of service operations using SWRL (Semantic Web Rule Language) rules and OWL ontologies to derive an extended finite state machine (EFSM) model. The authors propose various manual or automated techniques for generating test cases based on the EFSM model, with varying adequacy criteria, coverage and completeness. The generated EFSM consists of a single state and lacks information on explicit sequencing of operations (or conversation protocol) necessary to guide state-based testing.

Keum et al [11] provide a high-level, manual algorithm for constructing a multi-state EFSM model from plain WSDL specifications, with additional information provided by the user, who fills a WSDL analysis template. The described algorithm mainly focuses on the derivation of states, while little or no detail is given for obtaining the transitions and other EFSM elements.

In this paper we propose a method that utilises IOPE-based semantic descriptions of Web service operations that are encoded using RIF-PRD to generate a multi-state Stream X-machine model. In addition to the states and transitions that define a finite state machine, other constructs which are specific to SXMs, such as memory and processing functions, are also derived. The SXM testing strategy [9] allows for the derivation of a test set with completeness guarantees, making it possible to prove functional conformance of an implementation to the SXM model. The automated generation of test cases and their execution on a Web service under test are supported by an existing suite of tools.

## 3 IOPE-based Descriptions of Web Service Functionality

A popular approach for describing functional properties of stateful and conversational Web services is in terms of service inputs, outputs, preconditions and effects (IOPE) [20]. IO and PE descriptions are means to represent two different aspects of a service: (i) the information transformation that it produces through the inputs it consumes and the outputs it generates, and (ii) the state-wise conditions that need to hold before service operations can be invoked (preconditions) as well as the state changes that will be applied after the invocation (effects).

Descriptions of inputs and outputs allow service providers to explicate the semantics of the data vocabulary that the service operates on, independently of the service's behaviour. This type of specification can be provided for any kind of service, ranging from simple information-providing services to complex transactional services, and enables high-precision service discovery through semantic matchmaking, as well as run-time data mediation through semantic adapters. On the other hand, descriptions of preconditions and effects are only useful for Web services whose behaviour depends on the state of the world, and whose execution may cause changes to it. Examples of stateful Web services with side-effects could be an atomic ticket reservation service that consists of a single WSDL operation, or a bank account management service that comprises multiple WSDL operations. Our approach is focused on behavioural validation for services of the latter type, i.e. conversational Web services that maintain information about internal state between invocations of operations, and enforce a specific interaction protocol (choreography) with the service consumer.

Variations of the IOPE model are employed within all of the existing SWS frameworks. OWL-S process descriptions are based on an Input, Output, Precondition and Result (IOPR) model. A Result is associated to some Condition specifying the premises under which the result can occur, the corresponding Output, and the Effects to be produced. In descriptions of WSMO Goals and Capabilities a more fine-grained notion of Pre-conditions, Assumptions, Post-conditions, and Effects is employed to define state-wise constraints. In WSDL-S and SAWSDL, preconditions and effects can be associated with a Web service operation via a modelReference annotation on a WSDL portType, in the same way in which inputs and outputs are specified via modelReference annotations on WSDL operations or XSD elements.

In all cases, the representation of IOPE requires a combination of ontology language for encoding descriptions of inputs and outputs, and rule language for encoding preconditions and effects as logic-based expressions. Each of the previously mentioned SWS frameworks adopts a different approach to achieve this. OWL-S requires the representation of IO using OWL and allows encoding of logical expressions (treated as literals) for PE in a variety of rule languages, although it appears that only SWRL-like expressions have actually been supported in implemented OWL-S tools. In WSMO, Preconditions, Postconditions, Assumptions and Effects involve the specification of axioms as First Order Logic formulae that can be encoded in WSML dialects such as WSML-Rule or WSML-Flight. In WSDL-S and SAWSDL the format of the logical expressions is left to be defined by the modellers, and their semantic representation language of choice.

The kind of rules and rule language to be used for representing preconditions and effects depends on the intended semantics of the Web service's effects. We can generally distinguish among the use of deductive (inference) rules which have logical, non-monotonic semantics, and the use of reactive rules that carry operational, non-monotonic semantics. Reactive rules can be Event-Condition-Action (ECA) rules, which allow modelling changes to the state of the world as a reaction to an event, or Condition-Action (CA) rules, also known as production rules, which allow modelling state changes without reference to a particular event. A significant difference among deductive and reactive rules is found in the conclusion part of the rule. Conclusions of deductive rules contain only logical statements that should be true, whereas conclusions of reactive rules contain actions can add, delete, or modify facts in the knowledge base, and have other side-effects [21].

Our approach in this paper utilises production rules for specifying the behaviour of stateful and conversational Web services, and the rule language that we adopt is the Rule Interchange Format - Production Rule Dialect (RIF-PRD) [21]. RIF [22] is a forthcoming standard for the representation and exchange of rules that is currently under development by the W3C RIF Working Group, and is put forward as the next generation rule language for the Semantic Web. At the time of this writing RIF-PRD is still work in progress, but expected to become a W3C recommendation soon.

An important reason for using RIF-PRD is that the semantics of PRD rules can be specified as a labelled transition system [18], where states correspond to possible states of working memory and transitions correspond to actions taken by the rules to assert/retract terms. The fact that PRD has operational semantics makes the definition of an equivalence relationship between PRD and SXM specifications possible, and thus enables to prove the correctness of the transformation. Moreover, RIF-PRD can serve as a standard interface language between existing SWS formalisms and tools for derivation of SXM models and generation of test cases, thus bringing added value to existing SWS frameworks by allowing semantic descriptions to be reused for the purposes of testing and validation.

## Example Description of a Stateful Web Service

To provide an example of a stateful Web service whose functionality is modelled using the approach described above, consider the case of a service that allows performing elementary operations on a bank account. For simplicity, let us assume that the service interface comprises five operations: (i) open, (ii) deposit, (iii) withdraw, (iv) getBalance, and (v) close. 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 will result 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 will result in reducing the balance accordingly. A successful deposit or withdrawal will also result in having the updated balance returned to the client as part of the invocation response message. Finally, an account can be closed only if its balance is zero, and once closed cannot be re-activated. The listing below provides an encoding of preconditions and effects for this example Web service, using RIF-PRD presentation syntax.

```
Prefix(func http://www.w3.org/2007/rif-builtin-function#)
Prefix(pred http://www.w3.org/2007/rif-builtin-predicate#)
```

**(* wsdl:operation open *)**

```
Forall     ?account ?status ?balance ?request (
And     (  ?account#Account
           ?account[hasStatus->?status]
           ?account[hasBalance->?balance]
           ?request#OpenRequest )
If      (  External (pred:string-equal(?status "INITIAL") )
Then Do (  Retract (?account[hasStatus->?status])
           Assert (?account[hasStatus->"ACTIVE"])
           Retract (?account[hasBalance->?balance])
           Assert (?account[hasBalance->0])
           Retract (?request)
           (?response New(?response#OpenResponse))
           Assert (?response[hasMessage->"Account opened"]) ) )
```

**(* wsdl:operation close *)**

```
Forall     ?account ?status ?balance ?request (
And     (  ?account#Account
           ?account[hasStatus->?status]
           ?account[hasBalance->?balance]
           ?request#CloseRequest )
If And  (  External (pred:string-equal(?status "ACTIVE")
           External (pred:numeric-equal(?balance 0) )
Then Do (  Retract (?account[hasStatus->?status])
           Assert (?account[hasStatus->"CLOSED"])
           Retract (?request)
           (?response New(?response#CloseResponse))
           Assert (?response[hasMessage->"Account closed"]) ) )
```

**(* wsdl:operation getBalance *)**

```
Forall     ?account ?status ?balance ?request (
And     (  ?account#Account
           ?account[hasStatus->?status]
           ?account[hasBalance->?balance]
           ?request#GetBalanceRequest )
If      (  External (pred:string-equal(?status "ACTIVE") )
Then Do (  Retract (?request)
           (?response New(?response#GetBalanceResponse))
           Assert (?response[hasAmount->?balance) ) )
```

**(* wsdl:operation deposit *)**

```
Forall     ?account ?status ?balance ?request ?depositAmount (
And     (  ?account#Account
           ?account[hasStatus->?status]
           ?account[hasBalance->?balance]
           ?request#DepositRequest
           ?request[hasAmount->?depositAmount] )
If And  (  External (pred:string-equal(?status "ACTIVE")
           External (pred:numeric-greater-than(?depositAmount 0) )
Then Do (  Retract (?account[hasBalance->?balance])
           Assert (?account[hasBalance->External(func:numeric-add(?balance
           ?depositAmount))])
           Retract (?request)
           (?response New(?response#DepositResponse))
           (?newBalance ?account[hasBalance->?newBalance])
           Assert (?response[hasAmount->?newBalance) ) )
```

**(* wsdl:operation withdraw *)**

```
Forall     ?account ?status ?balance ?request ?withdrawAmount (
And     (  ?account#Account
           ?account[hasStatus->?status]
           ?account[hasBalance->?balance]
           ?request#WithdrawRequest
           ?request[hasAmount->?withdrawAmount] )
If And  (  External (pred:string-equal(?status "ACTIVE")
           External (pred:numeric-greater-than-or-equal(?balance
           ?withdrawAmount) )
Then Do (   Retract (?account[hasBalance->?balance])
           Assert(?account[hasBalance->External(func:numeric-subtract(?balance
           ?withdrawAmount))])
           Retract (?request)
           (?response New(?response#WithdrawResponse))
           (?newBalance ?account[hasBalance->?newBalance])
           Assert (?response[hasAmount->?newBalance) ) )
```

In addition to the rule-based description of service behaviour that is encoded in RIF-PRD, we assume the existence of an ontology-based description of the service's data model, encoded in OWL. The ontology is to be used for two purposes: i) specifying the structure of the service's inputs, outputs and internal state-related variables which are not visible on its interface, and ii) specifying the set of facts that hold when the service is found at its initial state (e.g. after original deployment or resetting). This information is utilised for the derivation of the SXM model, as described in section 5. The following table lists the OWL entities that the ontology would need to comprise:

**Table 1.** Ontology entities.

| | |
|---|---|
| **Classes (11)** | `Account, OpenRequest, OpenResponse, DepositRequest, DepositResponse, GetBalanceRequest, GetBalanceResponse, WithdrawRequest, WithdrawResponse, CloseRequest, CloseResponse` |
| **Datatype properties (4)** | `hasStatus (domain: Account, range: string enumeration {"INITIAL", "CLOSED", "FALSE"}), hasBalance (domain: Account, range: nonNegativeInteger), hasAmount (domain: {GetBalanceResponse, DepositRequest, DepositResponse, WithdrawRequest, WithdrawResponse}, range: nonNegativeInteger) hasMessage (domain: {OpenResponse, CloseResponse}, range: string)` |
| **Individuals (1)** | `acc1 (asserted Account individual, hasStatus: "INITIAL", hasBalance: 0)` |

Note that the RIF-PRD rules are not meant to refer to the ontology classes and properties in a direct way. Rather, they are meant to refer to in-memory representations of these entities which are created when the transformation procedure is initiated by reading-in the ontology contents. This hybrid integration approach has been chosen since RIF-PRD does not support importing of OWL ontologies in RIF-PRD documents and using ontology entities within rules, as is the case with RIF-BLD (Basic Logic Dialect). The reason for this is the incompatible semantics of OWL and RIF-PRD; OWL has monotonic model theoretic semantics, while production rules have non-monotonic operational semantics (due to the actions performed).

To complete the description of the example Web service, we also need a way to associate each of the production rules with the corresponding WSDL operation whose behaviour is meant to be described. This association can be specified by placing SAWSDL modelReference annotations on the corresponding wsdl:operation elements, and adopting a convention for identifying and referencing different rules within a RIF XML document (e.g. through XPointer fragment identifiers). Semantic annotations on the WSDL document can also serve for specifying the indirect linkage among the rule-based description of service behaviour and the ontology-based description of the service data model, which needs to be known for the SXM derivation. A complete description of an SAWSDL-based approach for linking the WSDL, RIF, and OWL artefacts is however beyond the scope of this paper.

## 4 Stream X-machines and Web Service Modelling

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 [6]. 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.

Apart from being formal as well as proven to possess the computational power of Turing machines [9], SXMs offer a highly effective testing method for verifying the conformance of a system's implementation against a specification. The SXM testing method, which is a generalization of the W-method [3], is guaranteed to generate a complete, finite set of test cases that can reveal all inconsistencies among an SXM specification and an implementation under test [10]. More details about the derivation of the test sequences can be found in [5].

Parallels can be drawn among a stateful Web service and a Stream X-machine, since they both accept inputs and produce outputs, while performing specific actions and moving from one internal state to another. SXM inputs and outputs correspond to SOAP request and response messages. SXM processing functions correspond to Web service operation invocations in specific contexts (an operation invocation may map to more than one processing functions because of the potentially different input parameters and the different state). In [5] the process of modelling and test generation for a stateful Web service are demonstrated in more detail, while in [12] we present an approach that integrates SXM-based modelling and test generation for extending a service registry with functional testing and behavioural verification capabilities.

## 5 Derivation of an SXM Model from an IOPE Description

This section describes the steps of the transformation of IOPE-based descriptions of Web service behaviour to SXM models. The example of the Account Web service is used throughout, for illustration purposes.

**Identifying State Variables.** All properties $p_i$ of frame atomic formulas `t[`$p_1$`->`$v_1$ … $p_n$`->`$v_n$`]` that appear in the action part of all rules are identified as state variables. Excluded are the frames associated to the reserved objects `?request` and `?response`, which represent inputs and outputs, respectively.

In the account service example two state variables are identified: `hasStatus` and `hasBalance`.

**Partition Analysis of State Variables.** The domains of the state variables are determined by consulting their respective *ranges* in the *datatype properties* section of the OWL ontology. Then, the domain of each state variable $p_i$ is partitioned based on the *preconditions* of the production rules. For each variable $v_i$ bound to a property $p_i$, which is a state variable, all formulas in the precondition of the rules are evaluated.

In the account example, the domains of the state variables are:

`hasStatus ::= {INITIAL, ACTIVE, CLOSED}` and `hasBalance::= 0…∞`.

The preconditions may restrict the state variables to actual values or ranges of values. In the rules of the account service example, the variables `?status` and `?balance` bound to the state variables `hasStatus` and `hasBalance` are evaluated in the following conditions:

```
hasStatus:
    • ?status = INITIAL
    • ?status = ACTIVE
hasBalance:
    • ?balance = 0
    • ?balance ≥ ?withdrawAmount AND ?withdrawAmount > 0
```

By application of naive arithmetic rules on the last condition listed above, it can be deduced that `?balance>0`. It should be noted that mathematical deductions like this can be hard to automate, and would require manual intervention. After examining the domains of the state variables and inferring the complements, the final partitions are:

```
hasStatus: {INITIAL}, {ACTIVE}, {CLOSED}
hasBalance: {0}, {x | x>0}
```

**Identifying Preliminary States.** The preliminary state space is defined as the product of the state variable partitions. The initial state $q_0$ is determined from the initial values of the respective state variables, which are specified in an individual in the OWL ontology.

In our example there are six preliminary states, which are labelled as `INITIAL_0`, `INITIAL_>0`, `ACTIVE_0`, `ACTIVE_>0`, `CLOSED_0`, and `CLOSED_>0`.

In the account specification the initial value for `hasStatus is INITIAL`, and for `hasBalance` is zero. Therefore state `INITIAL_0` is the initial state.

**Determining Inputs and Outputs.** An input (output) is defined for each service operation request (response) specified in the RIF rules. Inputs appear at the bindings and outputs at the action parts of the rules.

The inputs in the example are:

```
OpenRequest()
GetBalanceRequest()
DepositRequest(depositAmount)
WithdrawRequest(withdrawAmount)
CloseRequest()
```

The outputs are:

```
OpenResponse()
GetBalanceResponse(balance)
DepositResponse(newBalance)
WithdrawResponse(newBalance)
CloseResponse()
```

**Determining Transition Pre-States.** An *input* is *accepted* at a state (pre-state) iff the preconditions are satisfied at the pre-state. In that case the input triggers a *transition* of the Stream X-machine from the *pre-state* to another state (the *post-state*). For each input (service operation invocation) and each state, it is determined whether the input is accepted at that state.

For example, the preconditions of the rule for input `OpenRequest` are `(hasStatus=INITIAL)`, so they are satisfied at states `INITIAL_0 and INITIAL_>0`. On the other hand, input `WithdrawRequest` is a different case. By applying the previously mentioned mathematical deduction we can infer the following:

```
hasBalance≥withdrawAmount AND withdrawAmount>0 → hasBalance>0
```

Therefore, the input `WithdrawRequest` is only accepted at pre-state `ACTIVE_>0`. The inputs and the pre-states at which they trigger a transition are as follows:

```
OpenRequest: {INITIAL_0, INITIAL_>0}
GetBalanceRequest: {ACTIVE_0, ACTIVE_>0}
DepositRequest: {ACTIVE_0, ACTIVE_>0}
WithdrawRequest: {ACTIVE_>0}
CloseRequest: {ACTIVE_0}
```

**State Merging.** Preliminary states in which the same set of inputs is accepted are merged, since these states cannot be distinguished.

In our example, in both states `INITIAL_0` and `INITIAL_>0` the set of inputs `{OpenRequest}` is accepted. Therefore, those two states are merged into one. The resulting states are:

```
INITIAL: (hasStatus=INITIAL, hasBalance=*)
ACTIVE_0: (hasStatus=ACTIVE, hasBalance=0)
ACTIVE_>0: (hasStatus=ACTIVE, hasBalance>0)
CLOSED:  (hasStatus=CLOSED, hasBalance=*)
```

where * denotes any value.

**Determining Transition Post-States.** For each state-input pair identified in a previous step, we determine the possible transition destinations, or post-states, by applying the effects of the invoked operation on the pre-state. Isolated states (there does not exist any sequence of transitions starting from $q_0$ and ending in that state) are removed from the model.

In our example, applying these effects potentially involves reasoning with standard operations, such as the mathematical operations of *addition* and *subtraction*. Accepting input `OpenRequest` at pre-state `INITIAL: (hasStatus=INITIAL)` and applying the effects (`hasStatus` ← `ACTIVE` and `hasBalance` ← `0`) results in a post-state where `hasStatus=ACTIVE` and `hasBalance=0`. This post-state matches exactly with the state `ACTIVE_0`. On the other hand, applying the effects of accepting the input `WithdrawRequest`, and appealing to mathematical rules for subtraction, there are two possible post-states (`ACTIVE_0` and `ACTIVE_>0`), depending on the amount withdrawn. The two different transitions start from the same state, so they are labelled by different processing functions, for instance, `WithdrawRequest1` and `WithdrawRequest2`. Using the same approach for the rest of the state-input pairs, the resulting transitions are as in the following table.

**Table 3.** Transition pre-states and post-states.

| Input | Pre-State | Effects | Post-State |
|---|---|---|---|
| OpenRequest | INITIAL | hasStatus ← ACTIVE<br>hasBalance ← 0 | ACTIVE_0 |
| GetBalanceRequest | ACTIVE_0 | - | ACTIVE_>0 |
| | ACTIVE_>0 | - | ACTIVE_>0 |
| DepositRequest | ACTIVE_0 | hasBalance ← hasBalance+depositAmount | ACTIVE_>0 |
| | ACTIVE_>0 | hasBalance ← hasBalance+depositAmount | ACTIVE_>0 |
| WithdrawRequest | ACTIVE_>0 | hasBalance ← hasBalance-withdrawAmount | ACTIVE_0 |
| | ACTIVE_>0 | hasBalance ← hasBalance-withdrawAmount | ACTIVE_>0 |
| CloseRequest | ACTIVE | hasStatus ← CLOSED | CLOSED |

The final states and transitions define the *associated finite automaton* of the Stream X-machine, as illustrated in Fig. 1.

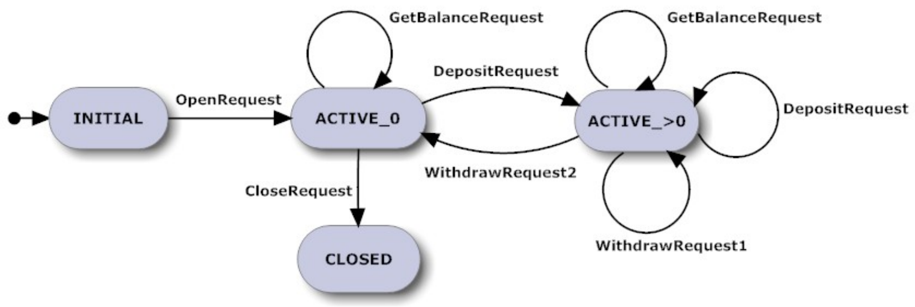

**Fig. 1.** State diagram of the SXM model that is the result of the transformation.

**Determining Memory.** The memory consists of all the state variables, except for those whose all possible values were covered by the partitions identified during partition analysis.

In the account service example, the enumerated state variable `hasStatus` was partitioned into all its possible values. Therefore, only state variable `hasBalance` is put in the memory. The initial memory $m_0$ is the initial value of `hasBalance`, i.e. $m_0 = (0)$.

**Determining Guard Conditions for Processing Functions.** For each state-input pair where the input triggers a unique transition, the guard conditions are the same as the corresponding rule preconditions; any predicates already satisfied in the pre-state are omitted.

For instance, when attempting to accept input `OpenRequest` at state `INITIAL`, all preconditions are already satisfied at the pre-state, so the guard condition is empty.

For each state-input pair where the input triggers more than one transition, the preconditions have to be split into mutually disjoint guard conditions, one for each processing function. For each transition, starting from its post-state, the effects are "undone" (i.e. the updates of the effects are inversed) and it is derived what should hold at the pre-state.

This is the case with input `WithdrawRequest(withdrawAmount)`. The only predicates which are not satisfied in the pre-state are `(hasBalance≥withdrawAmount AND withdrawAmount>0)`. In the post-state of `WithdrawRequest1`, `hasBalance>0`, while in the post-state of `WithdrawRequest2`, `hasBalance=0`. The effect of accepting the input `WithdrawRequest` is: `newBalance = hasBalance – withdrawAmount`. Since the value of updated balance in the post-state of `WithdrawRequest1` is greater than zero, then:

```
(hasBalance - withdrawAmount)>0 → hasBalance>withdrawAmount
```

Thus, the guard condition of `WithdrawRequest1` is (`hasBalance>withdrawAmount AND withdrawAmount>0)`. Similarly it can be deduced that the guard condition of `WithdrawRequest2` is `(hasBalance=withdrawAmount AND withdrawAmount>0)`.

**Determining Memory Updates for Processing Functions.** The memory updates of each processing function consist only of the effects which update the *memory variables*.

For example, the effect of the rule for `CloseRequest` does not involve any memory variables, thus the memory update function is empty. Similarly, for input `DepositRequest` the memory update function is `hasBalance ← hasBalance+depositAmount`, which is the only action affecting memory variable `hasBalance`.

## 6 Generation of Test Cases

The main benefit of obtaining a Stream X-machine model of the Web service is the ability to automatically generate complete functional test sets. This formalism is associated with a complete testing strategy which under certain assumptions [9] is proven to find all faults in the implementation. In addition, the SXM model can be animated and thus serve as a test oracle for predicting the service's outputs, which are compared with the real outputs returned by the service implementation.

Test generation starts by applying the W-method [3] on the associated *finite automaton* of the SXM, where processing functions are considered as simple inputs.

As a result, the test set *X* for the associated finite automaton consists of sequences of processing functions and 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 automaton, and *k* is an estimate of maximum path length between redundant states in the implementation. 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 next step is to convert the sequences of processing functions to sequences of inputs. This is achieved by the fundamental test function as described in [9].

The above test generation procedure is supported by a suite of tools that we have developed in Java. The test generation tool takes as input an XML representation of the SXM model and automatically produces sequences of abstract input and expected output pairs. Another tool converts the abstract test cases to JUnit test cases. In order to execute the generated JUnit test cases on the Web service under test, a Java client stub is required as an adapter. We use the open source WSDL2Java API provided by Apache Axis2 to automatically generate a client stub, which can invoke Web service operations when its Java methods are called.

In order to illustrate the generation of test cases for the account example, we have implemented and deployed a Web service that behaves according to the rule-based description provided in section 3. We have further created an SXM model based on the prescribed behaviour, generated test cases, and executed them on the deployed Web service. The table below lists the number of generated test sequences for different values of *k*, and the time taken to execute them. It should be noted that in all cases the time required for test generation was negligible (less than one second).

**Table 3.** Test sequences and execution times for different testing configurations.

| **Value of *k*** | **No. of test sequences** | **Execution time** |
|---|---|---|
| $k = 0$ | 25 | 3.9 s |
| $k = 1$ | 72 | 11.5 s |
| $k = 2$ | 230 | 45.2 s |

## 7 Conclusions and Outlook

Despite the significant amount of research interest in the area of Semantic Web Service technologies, and the numerous results already contributed towards automating service discovery, composition and mediation, the possibility of leveraging SWS descriptions for automated generation of test cases and behavioural validation has remained largely unexplored. In this paper we propose the use of Stream X-machines as a powerful behavioural modelling formalism for Web services, which can facilitate automated generation of test cases with completeness guarantees.

The test cases can be used for performing validation of Web services through automated functional testing (in a black-box manner). We have described a method for deriving an SXM model from an IOPE-based description of a Web service's functionality, and provided an example of encoding a stateful Web service's IOPE using RIF-PRD and OWL, to subsequently serve as input to the transformation.

The particular strengths of the presented approach can be summarised in three points. Firstly, a significant advantage of SXMs compared to other formalisms, is in the strength of their associated testing method, which can be used for proving functional equivalence among a specification and an implementation under test. This brings a significant advantage over other approaches which are not based on a formal theory of testing, or can only provide guarantees for weaker notions of equivalence. Secondly, the method of transforming an IOPE-based description of a Web service that has been encoded in RIF-PRD and OWL, into an SXM specification, makes our proposed approach generic and framework-independent. Given a service description compliant with some SWS framework, and a method of transforming such a description to one that is based on our presented modelling conventions, the derivation of an SXM model and generation of test cases will be possible. This is in contrast to other approaches for test case generation which are bound to specific SWS frameworks (e.g. OWL-S). Thirdly, the tasks of test case generation and test execution are readily supported by existing tools.

As future work we intend to investigate certain aspects of the SXM derivation method in greater detail, especially those steps that appear to be most challenging to automate, such as the partitioning of domains for state variables. We will further focus on the formalisation of the algorithm, considering complexity and decidability properties, and providing a proof of equivalence among the original rule-based specification and the derived SXM model. We will finally work towards the implementation of tools that automate the transformation, as well as experimental validation and collection of empirical evidence concerning the effectiveness of the overall approach.

## References

1. Akkiraju, R., Farrell, J., Miller, J., Nagarajan, M., Schmidt, M.T., Sheth, A., Verma, K.: Web Service Semantics - WSDL-S. W3C Member Submission (2005)
2. Bertolino, A., Frantzen, I., Polini, A., Tretmans, J.: Audition of Web Services for Testing Conformance to Open Specified Protocols. Architecting Systems with Trustworthy Components. LNCS, vol. 3938, pp. 1--25, Springer, Heidelberg (2006)
3. Chow, T.S.: Testing Software Design Modelled by Finite State Machines. IEEE Transactions on Software Engineering, vol. 4, pp. 178--187 (1978)
4. de Bruijn, J.d., Bussler, C., Domingue, J., Fensel, D., Hepp, M., Keller, U., Kifer, M., Konig-Ries, B., Kopecky, J., Lara, R., Lausen, H., Oren, E., Polleres, A., Roman, D., Scicluna, J., Stollberg, M.: Web Service Modeling Ontology (WSMO). W3C Member Submission (2005)
5. Dranidis, D., Kourtesis, D., Ramollari, E.: Formal Verification of Web Service Behavioural Conformance through Testing. Annals of Mathematics, Computing & Teleinformatics (AMCT), vol. 1, no. 5, pp. 36--43 (2007).

6. Eilenberg, S.: Automata, Languages and Machines, Volume A. Academic Press, New York (1974)
7. Farrell, J., Lausen, H. (eds.).: Semantic Annotations for WSDL and XML Schema. W3C Recommendation (2007)
8. Heckel, R., Mariani, L.: Automatic Conformance Testing of Web Services. In: Cerioli, M. (Ed.) FASE 2005, LNCS, vol. 3442, pp. 34--48, Springer Heidelberg (2005)
9. Holcombe, M., Ipate, F.: Correct Systems: Building Business Process Solutions. Springer Verlag, Berlin (1998)
10. Ipate, F., Holcombe, M.: An Integration Testing Method that is Proved to Find All Faults. International Journal of Computer Mathematics, vol. 63, pp. 159--178 (1997)
11. Keum, C., Kang, S., Ko, I.Y.: Generating Test Cases for Web Services using Extended Finite State Machine. In: Proceedings of the 18th IFIP International Conference on Testing Communicating Systems (TestCom 2006), pp. 103--117, Springer (2006)
12. Kourtesis, D., Ramollari, E., Dranidis, D., Paraskakis, I.: Discovery and Selection of Certified Web Services through Registry-Based Testing and Verification. In: Camarinha-Matos L. and Pickard W. (Eds.): Pervasive Collaborative Networks, IFIP volume 283, Springer Boston, pp. 473--482 (2008)
13. Martin, D., Domingue, J., Brodie, M.L., Leymann, F.: Semantic Web Services, Part 1. IEEE Intelligent Systems, vol. 22, no. 5, pp. 12--17 (2007)
14. Martin, D., Domingue, J., Sheth, A., Battle, S., Sycara, K., Fensel, D. Semantic Web Services, Part 2. IEEE Intelligent Systems, vol. 22, no. 6, pp. 8--15 (2007)
15. Martin, D., Burstein, M., Hobbs, J., Lassila, O., McDermott, D., McIlraith, S., Narayanan, S., Paolucci, M., Parsia, B., Payne, T., Sirin, E., Srinivasan, N., Sycara, K.: OWL-S: Semantic Markup for Web Services. W3C Member Submission (2004)
16. McGuinness, D.L., van Harmelen, F.: OWL Web Ontology Language Overview, W3C Recommendation (2004)
17. Narayanan, S., McIlraith, S. A.: Simulation, Verification and Automated Composition of Web Services. In: Proceedings of the 11th International Conference on the World Wide Web, pp. 77--88 (2002)
18. Plotkin, G.D.: A Structural Approach to Operational Semantics. Journal of Logic and Algebraic Programming, vol. 60-61, pp. 17--139 (2004)
19. Sinha, A. Paradkar, A.: Model-based Functional Conformance Testing of Web Services Operating on Persistent Data. In Proceedings of Workshop on Testing, Analysis and Verification of Web Services and Applications (TAV-WEB'06), pp. 17--22 (2006)
20. Urbieta, A., Azketa, E., Gomez, I., Parra, J., Arana, N.: Analysis of Effects- and Preconditions-Based Service Representation in Ubiquitous Computing Environments. In: Proceedings of the 2008 IEEE International Conference on Semantic Computing, pp. 378--385 (2008)
21. W3C RIF Production Rule Dialect (RIF-PRD) http://www.w3.org/TR/rif-prd/
22. W3C Rule Interchange Format (RIF) Working Group http://www.w3.org/2005/rules/
23. Wang, Y., Bai, X., Li, J., Huang, R.: Ontology-Based Test Case Generation for Testing Web Services. In: Proceedings of Eighth International Symposium on Autonomous Decentralized Systems (ISADS '07), pp.43--50 (2007)