Article:
Kourtesis, Dimitrios, Ramollari, Ervin, Dranidis, Dimitris et al. (1 more author) (2010) Increased reliability in SOA environments through registry-based conformance testing of Web services. Production Planning & Control: The Management of Operations. Special Issue on Engagement in Collaborative Networks., 21 (2). pp. 130-144. ISSN 1366-5871

https://doi.org/10.1080/09537280903441922

The following paper is a post-print (final draft post-refereeing) of:

Kourtesis, D., Ramollari, E., Dranidis, D., & Paraskakis, I. (2010). Increased Reliability in SOA Environments through Registry-Based Conformance Testing of Web Services. *Engagement in Collaborative Networks. Special Issue in International Journal of Production Planning & Control: The Management of Operations (JPPC)*, 21(2), 130–144

Publisher's version is available at:

http://www.informaworld.com/smpp/content~content=a919639247

Increased Reliability in SOA Environments through Registry-Based Conformance Testing of Web Services

D. Kourtesis[1], E. Ramollari[1], D. Dranidis[1,2], I. Paraskakis[1]

[1] *South East European Research Centre (SEERC), Research Centre of the University of Sheffield and CITY College*
24 Proxenou Koromila Str., 54622, Thessaloniki, Greece
Tel: +30 2310 253477, Fax: +30 2310 234205,
Email: dkourtesis@seerc.org, erramollari@seerc.org, iparaskakis@seerc.org

[2] *Computer Science Department, CITY College, International Faculty of the University of Sheffield*
3 Leontos Sofou Str., 54626, Thessaloniki, Greece
Tel: +30 2310 528450, Fax: +30 2310 282384,
Email: dranidis@city.academic.gr

Abstract: Organisations wishing to engage in industrial collaborative networks will typically seek some guarantees concerning the reliability of their prospective partners before committing to cooperation. Evaluating reliability can encompass several aspects, but one of the most crucial things to consider from a cooperation perspective is whether the software systems that support the business processes of some collaborator actually behave as expected. For organisations that rely on a service-oriented computing infrastructure, this amounts to checking whether the functionality of the respective services is conformant to a given behavioural specification. Today's state of the art lacks standardised methods for creating behavioural specifications of Web services, and also lacks tools for automating the process of behavioural conformance checking through testing. This paper presents a concrete method for creating formal specifications of Web service behaviour and utilising them within service registries for automated testing of service implementations in order to verify and certify their conformance.

Keywords: Web services, registry, behavioural conformance, testing

1. Introduction

Reliability is a fundamental prerequisite for establishing effective cooperation among any two entities. For this reason, organisations that wish to engage in industrial collaborative networks will typically seek guarantees concerning the reliability of their prospective partners before committing to cooperation. In cases of collaborative networks where some organisation acts as a central authority for coordination and brokerage, such as Virtual Organisation Breeding Environments (VBEs) (Camarinha-Matos and Afsarmanesh 2005), the task of reliability evaluation for prospective or existing members of the network can be part of the coordinator's overall management responsibilities. By delegating the task of reliability assessment to an authoritative and accountable entity, network members can be reassured as to the level of trustworthiness of their prospective business partners in a consistent and transparent manner. This function can promote the establishment of a proper balance of trust levels among organisations in the network, which is critical for the effectiveness of its operation (Msanjila and Afsarmanesh 2007, Camarinha-Matos 2007).

Kourtesis, D., Ramollari, E., Dranidis, D., & Paraskakis, I. (2010). Increased Reliability in SOA Environments through Registry-Based Conformance Testing of Web Services. *Engagement in Collaborative Networks. Special Issue in International Journal of Production Planning & Control: The Management of Operations (JPPC)*, 21(2), 130–144

Evaluating reliability can encompass several different perspectives and several layers of abstraction which incorporate both business-focused and technology-focused dimensions. However, under all circumstances, one of the most crucial aspects to consider from the perspective of cooperation and interoperability is whether the software systems that support the business processes of some collaborator actually behave as expected. Expectations concerning the way in which the systems of a prospective business partner should function may arise for several different reasons.

Firstly, a business partner may be required to conform to some specific industry standard that prescribes a particular interaction protocol for the parties engaged in a business process. An example could be conformance to information interchange description standards such as ebXML-MSS (Message Service Specification), which provide vendor-independent means for exchanging messages among business information systems for industry verticals (such as automotive or finance) and cross-industry collaborations. Another example could be business process description standards such as ebXML-BPSS (Business Process Specification Schema) and RosettaNet PIP (Partner Interface Process), which focus in providing company-independent and generic definitions of how business collaborations can be realised electronically (e.g. stock replenishment).

Secondly, the software systems of a business partner may be required to adhere to some specific behaviour as explicated in a service provision contract. The contract can represent an agreement over functional or non-functional aspects of service delivery which is binding for the interactions of the business partner with a specific member of the network, or with every member of the network. The contract may have resulted from a bilateral negotiation among the partner and some network peer (i.e. a member or the coordinator of the collaborative network), or may represent a public statement and commitment on behalf of the partner concerning the way in which services are to be delivered, as a means to promote interoperability.

Moreover, expectations concerning the behaviour of a business partner's systems may arise as the result of various imperatives for compliance with regard to legal, fiscal or trading standards and regulations. The policy compliance and IT management literature features an abundance of standards that could serve as examples. Relevant compliance guidelines that affect enterprise IT include laws on the protection of personal information (e.g. US Personal Data Privacy & Security Act of 2005) which define standards in business practices to ensure data privacy and security, or legislations like Sarbanes-Oxley Act (SOX) and Euro-SOX which define rules that affect the management of electronic records.

For collaborative network infrastructures based on a Service-Oriented Architecture and Web service technology standards, the task of verifying that the software systems of some collaborator operate as expected requires testing that the functionality of the respective Web services is conformant to a given behavioural specification. A proof of conformance among these two would constitute a measurable trust element (Msanjila and Afsarmanesh 2007b) towards facts-based assessment of trust levels for organisations participating in a network. For functional conformance checking to be possible, two requirements are set:

- A method is required for creating platform-independent specifications of Web service functionality. The method should be expressive enough to allow representing the behaviour of non-trivial Web services that fulfil arbitrarily complex business processes and may therefore need to be conversational and stateful. Moreover, the method should be formal, so as to allow generating exhaustive test cases that suffice for proving whether a Web service implementation is functionally equivalent to its respective specification, or not.
- Tool support is required for exploiting the behavioural specifications created with the abovementioned method, in order to generate test cases and perform the actual tests against deployed Web services. The tools should be advanced enough to allow automating the

procedures of test case generation and functional testing to the greatest extent possible. By integrating such tools in the infrastructure of authoritative entities within collaborative networks such as coordinators and brokers, effective and efficient reliability evaluation could be made possible.

Despite the existence of several standards around Web service technologies, a standardised method for creating formal behavioural specifications of conversational and stateful Web services has yet failed to emerge. A number of individual approaches have been proposed in the literature for modelling the behaviour of such services in a way that would allow generating test cases, but none of them provides guarantees for completeness and for being able to verify functional equivalence. In the absence of a suitable method, today's state of the art also lacks appropriate tool support for automating the process of behavioural conformance verification through testing. Today's commercial solutions for Web service testing and verification are primarily manual and demand a significant investment of resources on behalf of the tester. For several application areas, this can be an important barrier to adoption.

In this paper we present a concrete approach that is aimed at overcoming these deficiencies. First of all, we propose the use of Stream X-machines (Laycock 1993, Holcombe and Ipate 1998) as a formal modelling method for constructing behavioural specifications of complex Web services which are stateful and conversational. Apart from their expressive power, a significant advantage of Stream X-machines (SXMs) compared to other formalisms for modelling of external system behaviour is in their associated method for test case generation and verification. The sequences of test cases that can be generated from a SXM model can be proven to be exhaustive and able to reveal all inconsistencies between a given specification and an implementation under test (Dranidis et al. 2007). Another major advantage of using SXMs for Web service behaviour verification is the availability of a comprehensive suite of tools for automated generation of test cases and their execution on deployed Web services.

On the grounds of the availability of this mature method and supporting tool suite, we put forward an approach for augmenting the management infrastructure of authoritative entities in collaborative networks, such as brokers and coordinators in Virtual Organisation Breeding Environments. We propose to extend the functionality of Web service registries that are part of the management infrastructure of such entities, with capabilities for functional testing and behavioural verification.

We envisage the development of enhanced Web service registries that are able to process a Web service's SXM behavioural specification at the time of the service's publication, generate test cases from the model, execute the tests, and based on the responses of the service evaluate whether it is functionally equivalent to the associated specification, in order to provide certification. Successful certification, as a result of successful conformance verification of service-based systems against their specifications, is perceived as an objective measure and indicator of credibility for the service-provisioning organisation, and as such, constitutes a step towards building relationships of trust within a collaborative network. Beyond the above, we envisage an approach that also promotes efficiency after the phases of certification and Web service discovery, during service selection. The SXM specification and the generated test cases are to be used not only for verification at the registry's side, but also for validation at the requestor's side. Specifically, the tool suite that supports the approach presented in this paper enables service requestors to "simulate" the behaviour of a Web service and evaluate its usefulness without really testing it, but rather, by executing the test cases generated earlier by the registry against the service's SXM specification and inspecting the outputs that are generated by the model, using a dedicated SXM animator tool.

This paper represents an extension to our recent work as reported in (Kourtesis et al. 2008), and is organised as follows. Section 2 discusses related work in the domain of model-based Web service

testing and verification. Section 3 provides an overview of the Stream X-machine modelling formalism that is the basis of our approach. Section 4 provides a detailed walkthrough of our proposed method for Web service behaviour modelling and testing using a case study from the domain of manufacturing supply networks. Section 5 provides an overview of the whole approach for registry-based testing, verification and certification of Web services, presenting the approach from the perspectives of the provider, the registry and the requestor, and emphasising on their associated activities. Section 6 concludes the paper by summarising the main points of the presented work and outlining objectives for future research.

2. Related Work

A number of approaches have been proposed in recent years for the verification of Web services by employing model-based testing. In (Sinha and Paradkar 2006) a method is proposed for annotating a WSDL document with concepts from an OWL ontology representing inputs, outputs, preconditions and effects, and automatically translating the resulting WSDL-S specification into a semantically-equivalent extended Finite State Machine (EFSM) model. A set of manual or automated techniques for generating test cases based on the EFSM model is also provided. The techniques vary in terms of adequacy criteria, coverage and completeness.

The use of an EFSM modelling formalism for describing the dynamic behaviour of a Web service is also proposed in (Keum et al. 2006), where a manual procedure is suggested for deriving the EFSM model from a WSDL description. The proposed EFSM model is an FSM extended with memory, predicate conditions and computing blocks for state transitions. With proper tool support the EFSM model can be used for automatically generating Web service test cases with increased test coverage that includes both control flow and data flow. The authors provide experimental results showing that their method has the potential to find more faults compared to other methods, but notably without completeness guarantees.

The number of research works proposing the incorporation of Web service model-based testing and verification functionality in service registries is rather limited. The addition of a lightweight verification mechanism to UDDI service registries was first proposed in (Tsai et al. 2003). The key idea was to attach so-called “test scripts” to Web service specifications for both service registry and service consumers to use. Before publishing a service advertisement at the service registry or before consuming a service the associated test scripts could be used to test the actual service and verify its behaviour. The proposed approach is very abstract and does not prescribe the use of a specific formal or informal method of representing service behaviour, nor one for generating the test scripts.

In (Bertolino et al. 2005) the authors propose a framework with an enhanced UDDI registry that generates test cases for Web services, executes them, and monitors the interactions between the service under test and other services already registered with the framework in order to verify conformance to the published specification. Emphasis is placed on verifying that a Web service is interoperable with other registered services, and the framework is called an “audition framework” in the sense that a Web service undergoes a monitored trial before being admitted. The authors suggest that the behavioural service specification should be expressed as a UML 2.0 Protocol State Machine (PSM) diagram that can be semi-automatically transformed into a Symbolic Transition System (STS) on which existing automated test generation methods can be readily applied. The utilisation of the proposed behavioural specification formalism for matchmaking among service advertisements and requests is left undefined. Discovery is assumed to be supported by the typical means available in UDDI, i.e. keyword-based search and categorisation.

In (Heckel and Mariani 2005) the authors propose a “high-quality service discovery” approach that incorporates automatic testing and verification of Web Services before allowing their registration to the service registry. The authors propose Graph Transformation (GT) rules as the modelling formalism to be used for constructing behavioural service specifications. Conformance test cases are to be automatically generated from the provided specification and executed against the target Web Service. If the test is successfully passed, the service can be registered. Apart from testing and verification the GT-based service specifications can be also used for matchmaking among services and service requests that have been also expressed via GT rules. The proposed approach does not prescribe the use of UDDI or any other specific service registry specification as the technical infrastructure to support the approach.

A significant drawback in the above model-based verification approaches is that the test case derivation methods they employ cannot guarantee completeness in testing of the Web service implementations. In contrast, the Stream X-machine testing method on which our approach relies, is proven to generate a complete set of test cases that can reveal all inconsistencies among an implementation under test and an SXM specification (Ipate and Holcombe 1997). Moreover, a novel proposition in our approach is the use of the behavioural service specification by service requestors to perform validation after discovery, during the phase of service selection, through model animation, or even through model checking. Validation is an important utility for prospective service consumers, since it can assist them in selecting the most appropriate services from a list of candidates, regardless of the matchmaking and discovery method that was used to deliver this list.

3. 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 (Eilenberg 1974). 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. The benefit of adding the memory construct 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’s abstract control structure. 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 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, $\varphi : \Sigma \times M \rightarrow \Gamma \times M$;
- F is the next state partial function that given a state and a function from the type Φ, provides the next state, $F : Q \times \Phi \rightarrow Q$ (F is often described as a state transition diagram);
- 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 (Holcombe and Ipate 1998), SXMs offer a highly effective testing method for verifying the conformance of a system's implementation against a specification. Stream X-machine models can be represented in XMDL (X-Machine Definition Language), a special-purpose markup language introduced in (Kapeti and Kefalas 2000) or in XMDL-O, an object based extension of XMDL introduced in (Dranidis et al. 2005). XMDL-O enables an easier and more readable specification of Stream X-machines and it is the language that we are using in this paper. Additionally, a suite of supporting tools (JSXM) has been developed (Dranidis 2009) which can be used for the animation of SXM models and model-based automated test generation. Specifications in JSXM have an XML-based representation which facilitates native integration with Web technologies and related XML-based Web service standards. In the remainder of this paper we however utilise XMDL-O for illustration, because it is less verbose than the XML-based syntax and allows for easier understanding by the reader.s

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 invocation may map to more than one processing function). In addition, the modeller has to define the memory structure, not only as a substitute for internal state, but also to supply sample test data that can become part of the generated test sequences. 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.

4. Case study: Manufacturing Supply Network

4.1 Web service description

In order to illustrate our formal modelling and conformance testing approach, we use an example inspired from the domain of manufacturing supply networks, where a manufacturer orders new raw materials from a supplier partner. To perform this transaction, the manufacturer's production scheduling system interacts with the supplier's order processing system which is made available as a Web service. The transaction is performed in a number of steps and in accordance with a conversation protocol. The *SupplyOrder* Web service consists of the following operations: `login`, `logout`, `createOrder`, `cancelOrder`, `addItem`, `removeItem`, `getQuote`, `rejectOrder`, and `confirmOrder`, which can be called in sequences permissible by the conversation protocol. We have selected on purpose a Web service with complex behaviour and operating on complex data repositories, in order to be closer to the kinds of Web services that are expected to be found in the industry.

Before the manufacturing system can perform any action, it first has to be authenticated by invoking the `login` operation, with a request message containing a username and a password. The `logout` operation logs the manufacturer out of the system and ends the temporarily-created session. Normally, in accordance with the *CRUD* (create-read-update-delete) lifecycle of data objects, the manufacturer should be able to create a new order, and read, update, or delete an existing order. However, for simplicity, in this scenario we only model the creation of a new empty order with the `createOrder` operation. The manufacturer can populate the new supply order by adding items

specifying their id and requested quantities, through the repetitive invocation of the `addItem` operation. Order items can also be removed or the order cancelled altogether, after which the manufacturer has to create a new order. The `addItem` operation is fulfilled without checking for availability in the inventory, since this check is performed in the end when the manufacturer is ready to complete the supply order, by invoking the `getQuote` operation. The `getQuote` operation returns an order quotation (unless the order is empty), listing the items that are ordered, their availability and their prices. This gives the manufacturer the choice to proceed with the confirmation of the order, even if it is partially fulfilled (because some items are out of stock), or alternatively reject the order. The `getQuote` operation temporary locks the ordered items of the requested (or available) quantities in the inventory, so that no other client simultaneously accessing the system can order them until the current order is confirmed or rejected. Upon confirmation of the supply order, the item quantities that are fulfilled are subtracted from the inventory and the transaction ends.

4.2 Formal modelling with a SXM

The *SupplyOrder* Web service is a stateful service, since it maintains session state and the results of operation invocations are dependent on previous invocations. Figure 1 depicts the associated finite automaton of the SXM model created for the *SupplyOrder* Web service. It has to be noted that the transitions on the diagram are not labelled by Web service operations but by processing functions of the SXM. Although some of the operations, such as createOrder, are modelled as single processing functions, other operations, such as removeItem, are modelled by more than one processing functions that are triggered under different conditions.

For example, input `loginRequest` may trigger either a transition that leads to the `initial` state or a transition that leads to the `authenticated` state. Those two transitions have to be labelled by unique processing functions, e.g. `loginFailed` and `loginOK`. Similarly, input `getQuoteRequest` triggers three different transitions, `getQuoteEmpty`, when none of the items in the order are available; `getQuotePartial` when the items in the order are only partially available (the client can still proceed to confirming the order); and `getQuoteFulfilled`, when all of the items in the order are available and the order is completely fulfilled. Although the latter two transitions have the same initial and final states, they have been modelled separately to be distinguished as different cases during the test generation process.

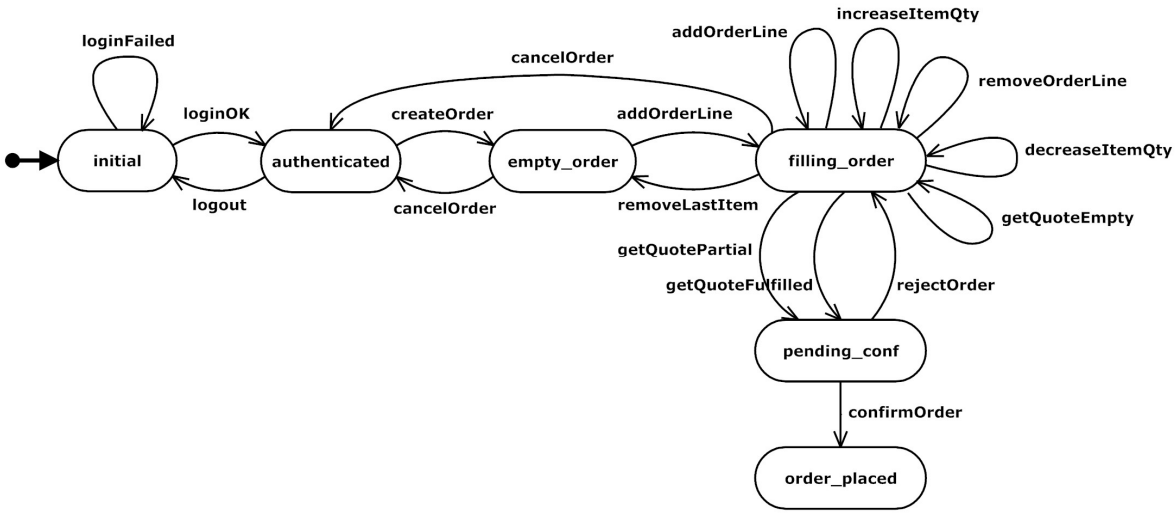

Figure 1 – Associated finite automaton for the *SupplyOrder* Web service SXM model

In the following, XMDL-O (Dranidis et al. 2005) code is used to illustrate parts of the specification, while the full specification is provided in the Appendix. The states and transitions of the SupplyOrder Stream X-machine model are defined in XMDL-O as follows:

```
#states = {initial, authenticated, empty_order, filling_order, pending_conf, order_placed}.
#init_state {initial}.

#transition (initial, loginFailed) = initial.
#transition (initial, loginOK) = authenticated.
#transition (authenticated, logout) = initial.
#transition (authenticated,createOrder) = empty_order.
#transition (empty_order,cancelOrder) = authenticated.
...
```

The memory of the SXM model is structured as a list of user accounts objects, a list of inventory item objects, and a list of order item objects. These lists are initialized to sample values, which are used only during the testing process. The memory and initial memory are defined in XMDL-O as follows:

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

#class InventoryItem {
 id: string,
 qty_in_stock: natural0,
 qty_on_hold: natural0,
}.

#class OrderItem {
 id: string,
 qty_requested: natural0,
 qty_reserved: natural0,
}.

#objects:
 account1: Account,
 accounts: set_of Account,
 inventoryItem1: InventoryItem,
 inventoryItem2: InventoryItem,
 inventory: set_of InventoryItem,
 order: set_of OrderItem.

#init_values:
 account1.username <- "usr1",
 account1.password <- "pwd1",
 inventoryItem1.id <- 1001,
 inventoryItem1.qty_in_stock <- 100,
 inventoryItem1.qty_on_hold <- 0,
 inventoryItem2.id <- 1002,
 inventoryItem2.qty_in_stock <- 50,
 inventoryItem2.qty_on_hold <- 0,
 accounts <- {account1},
 inventory <- {inventoryItem1, inventoryItem2},
 order <- emptySet.
```

XMDL-O also supports structured inputs, which are determined through events and their parameters. For example, a login request message comprising of a username and password is modelled as:

```
#event loginRequest(usr:string, pw:string).
```

The outputs are modelled as abstract messages and define an enumerated type. In the SupplyOrder example the outputs are:

```
#output (messages).

#type messages = {loginOk, loginFailed, loggedOut, orderCreated, orderCanceled, itemAdded,
itemQtyIncreased, itemRemoved, removeFailed, itemQtyDecreased, lastItemRemoved, quoteEmpty,
quotePartial, quoteFulfilled, orderRejected, orderConfirmed}.
```

The type Φ of the Stream X-machine is the set of processing functions labelling transitions. Each processing function receives input symbols and reads memory values, and produces output symbols while modifying memory values. In XMDL-O the guard conditions are defined in the *if*-clause, while the memory updates in one or more *update*-clauses.

One of the simplest processing functions is `createOrder`, which is simply triggered by an input (`createOrderRequest()`) and produces an output (`orderCreated`). There is no guard condition or memory updates.

```
#fun createOrder( createOrderRequest() )=
  (orderCreated).
```

Processing function `loginOK` receives a complex input comprising of a username and a password. The list operation `select(condition, list)` returns those items in the list for which the condition is satisfied. The results are used to determine whether a user account with the provided username exists, and then the provided password is compared with the password of the found user account. Although there are no memory updates, an output is produced, and a state transition is triggered.

```
#fun loginOK( loginRequest(?usr,?pw) )=
  if not_isempty ?found and ?current_user.pw=?pw
  then
     (loginOk)
  where
     ?found <- select (usr=?usr, accounts) and
               ?current_user <- head(?found).
```

An example of a significantly complex processing function is `getQuoteFulfilled`, which returns an order quote to the Web service client, when all the order items of the requested quantities are available. For testing purposes we model the output as a simple message (`quoteFulfilled`). The guard condition triggering this processing function is highly complex, since it has to iterate through all order items and check their availability in the inventory. A compound conjunction on all the elements of the list is performed with the conjunction operation, which has the form:

`conjunction (fn ?x => (`*condition*`),` *list*`)`

where `?x` is a variable that binds to elements of the list *list* and *condition* is a Boolean expression involving element `?x`.

In addition, processing function `getQuoteFulfilled` performs updates on the inventory as well as on the order list. The quantities on hold are increased for each item in the inventory, and the increase is also recorded for each `OrderItem` in the `qty_reserved` attribute. These updates are specified in XMDL-O with the map list operation, which maps a list of elements into another list and returns the result. It has the form:

`map (fn ?x =>` *expression*, *list*`)`

where `?x` is a variable that binds to elements of the list *list.* The result of the mapping is a list of the values of the expressions after variable substitution with the elements of the list. In the inventory update, this mapping is conditional and f(?x) uses the syntax:

```
if condition then value1 else value2.
```

The complete XMDL-O code (with comments) for `getQuoteFulfilled` is as follows:

```
#fun getQuoteFulfilled( getQuoteRequest() )=
   /*
      if for each id in the order, the requested quantity is less than or equal to
      the quantity in stock minus the quantity on hold
       (ie. The product is available in the requested quantity)
   */
   if conjunction (fn ?id =>
      (?invline.qty_in_stock - ?invline.qty_on_hold ≥ ?line.qty_requested
         where
            ?line <- select(id=?id, order) and
            ?invline <- select(id=?id, inventory)
      ),
      ?order_ids)
        then
      (quoteFulfilled)

   update
   /*
      update the items in the inventory which are ordered;
      increase quantity on hold by requested quantity
   */
      inventory <- map
      (fn ?invitem =>
         (new InventoryItem(?invitem.id, ?invitem.qty_in_stock,
            ?invitem.qty_on_hold + ?increase )
            where
               ?orditem <- head(select(id=?invitem.id,orders) and
               ?increase <- (if ?invitem.id belongs ?order_ids
                  then ?orditem.qty_requested else 0)
      ),
      inventory)
   and
   /*
      update the lines in the order; set the quantity requested.
   */
      order <- map
      (fn orditem =>
          (new OrderItem(?orditem.id, ?orditem.qty_requested,
            ?orditem.qty_requested)
      ),
      order)
   where
      ?order_ids <- project(id, order).
```

For the complete specification of all processing functions, the reader can refer to the Appendix.

4.3 Test generation

The SXM testing method is a generalization of the W-method (Chow 1978) and works on the basis that both specification and implementation could be represented as Stream X-machines with the same type F (i.e. both specification and implementation have the same processing functions), where F satisfies two fundamental design for test conditions: (i) completeness with respect to memory – all processing functions can be exercised from any memory value using appropriate inputs, and (ii) output distinguishability – any two different processing functions will produce different outputs if applied on the same memory/input pair.

For our testing approach to be applicable, Web service operations must follow the request-response message exchange pattern, i.e. they must return a response message for every request message they receive by the consumer. This makes it possible to fulfil the condition for output distinguishability, and also enables the testing engine to understand which processing functions have been activated during an execution path based on the responses of the service.

The first step for test generation is to construct the test set of input sequences by applying the W-method 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 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. For example, the *W*, *S*, and *Φ* sets in the *SupplyOrder* Web service example are:

```
W = {<loginOK>, <createOrder>, <addOrderLine>, <getQuoteFulfilled>, <confirmOrder>}

S = {<>, <loginOK>, <loginOK, createOrder>, <loginOK, createOrder, addOrderLine>, <loginOK,
createOrder, addOrderLine, getQuoteFulfilled>, <loginOK, createOrder, addOrderLine,
getQuoteFulfilled, cofirmOrder>

Φ = {<loginOK>, <loginFailed>, <logout>, <createOrder>, <cancelOrder>, <addOrderLine>,
<increaseItemQty>, <removeOrderLine>, <decreaseItemQty>, <removeError>,
<removeLastOrderLine>, <getQuoteEmpty>, <getQuotePartial>, <getQuoteFulfilled>,
<rejectOrder>, <confirmOrder>}
```

For k=0, the resulting test set X is $S(\Phi \cup \{\epsilon\})W$. This test set consists of sequences of processing functions, which have to be converted to sequences of abstract inputs. This is achieved by the fundamental test function as described in (Holcombe and Ipate 1998). For example, for the sequence <loginOK, createOrder, addOrderLine, getQuoteFulfilled, confirmOrder>, the generated sequence of inputs is:

```
<loginRequest("usr1", "pwd1"), createOrderRequest(), addItemRequest("1001", 1),
getQuoteRequest(), confirmOrderRequest()>
```

Since the specification is not input-complete, some of the sequences are not realizable, so that they are left out.

For the XML-based representation of Stream X-machine models the described test-set generation process is automated by the JSXM tool (Dranidis 2009). The tool can be used to animate models, generate abstract XML test cases, and map the abstract test cases to JUnit test cases automatically. We have also utilised various libraries (such as Apache WSDL2Java) to automatically generate Java Web service client stubs that can invoke Web service operations by calling the stubs' Java methods. Therefore, by running the generated JUnit test cases on the client stubs we actually execute them on the Web services under test.

5. Overview of the Approach

The approach that we put forward in this paper for registry-based testing and certification of Web services involves all three types of stakeholders in a SOA environment, i.e. service providers, service registries, and service requestors (consumers). As depicted in figure 2, the role of each stakeholder is associated with a number of activities. In brief, we propose that the behaviour of a Web service should be formally modelled at the provider-side, in order to facilitate registry-side verification at the time of service publication and requestor-side validation at the time of service selection. In the following three sections we present an overview of the activities performed by each stakeholder in the scheme.

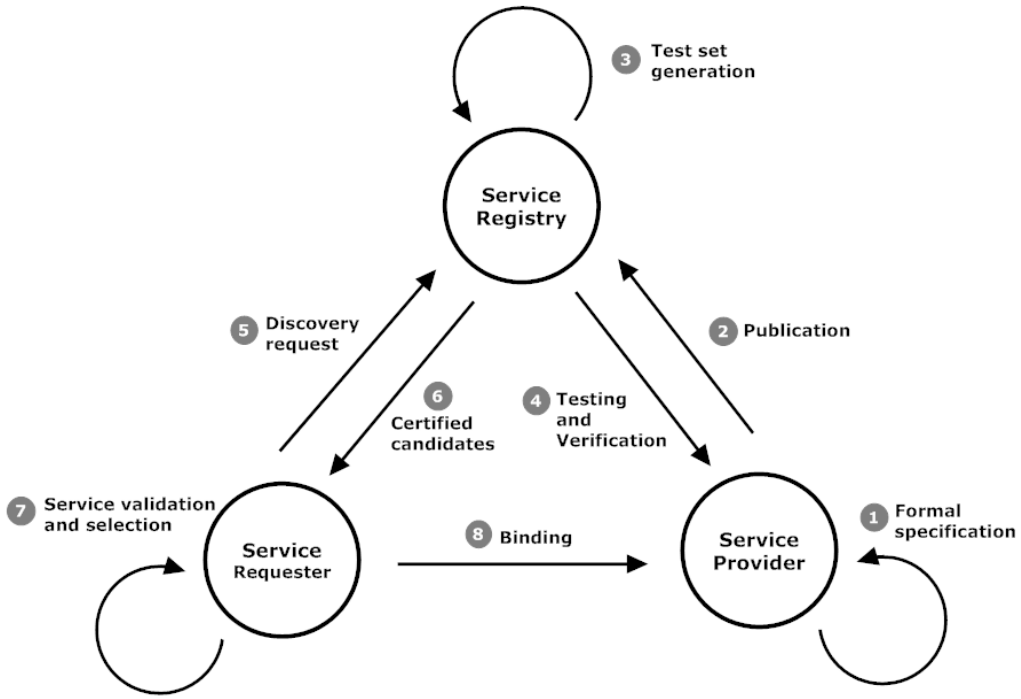

Figure 2 – Stakeholder roles and ordering of activities in the proposed approach

5.1 Construction of Web service behavioural specification

The objective of the service provider at this stage is to construct a formal model reflecting the behaviour of the service to be published (activity 1 in figure 2) using the Stream X-machine (SXM) formalism as described in sections 3 and 4. The SXM model is encoded in XMDL-O (or in the corresponding XML representation supported by JSXM) and stored in an external

document that must be subsequently “linked” with the service’s WSDL document. The association among the two document artefacts can be established by employing the SAWSDL (Semantic Annotations for WSDL) (Farrell and Lausen 2007) specification and its mechanism for annotating Web service descriptions with pointers to externally maintained semantically-rich specifications. In order to indicate the association between the two documents, an SAWSDL modelReference annotation pointing to the URL of the SXM specification document must be placed within the wsdl:portType definition of the service’s WSDL document.

The process of constructing an SXM model from a WSDL description can be automated to a great extent by describing Web service inputs, outputs, preconditions and effects (IOPE) using Semantic Web technologies, and then pointing to those semantic descriptions from within the WSDL document through SAWSDL annotations. Web service IOPE can be described through a combination of ontology language for representing operation inputs and outputs, and rule language for representing operation preconditions and effects as logical expressions.

The description of a method for representing Web service inputs and outputs in an OWL-DL ontology and pointing to them from within a WSDL document via SAWSDL annotations is provided in (Kourtesis and Paraskakis 2008). We have also developed a method for representing preconditions and effects using RIF-PRD (Rule Interchange Format - Production Rule Dialect) in conjunction with OWL-DL, and utilise both for deriving a Stream X-machine behavioural specification (Ramollari et al. 2009).

Modelling of IOPE semantics in the above manner would not only assist in increasing the automation of the SXM model construction process, but could also serve as a basis for performing behaviourally-aware service matchmaking for high-precision retrieval of services, thus extending the method and tools presented in (Kourtesis and Paraskakis 2008b).

Nevertheless, regardless of the method used to construct the SXM specification, manual or semi-automated one, as soon as the SXM model is completed and the WSDL document has been semantically annotated, the provider must submit it to the service registry for publication (activity 2).

5.2 Generation of test cases, testing, and certification

The objective of the service registry at this stage is to verify that the service implementation is functionally conformant to its associated specification, and if this holds, provide a certification for the service advertisement. All activities within the service registry are automated, and their ordering is as follows. Firstly, the registry processes the incoming SAWSDL description and creates a service advertisement with a status of pending certification. Secondly, the attached SXM specification is used for deriving a complete set of test cases that can reveal all inconsistencies in the service implementation to be verified (activity 3). Lastly, the executable tests are run by the registry’s SOAP testing engine and if the results are successful (i.e. if the produced outputs match the expected ones) the service advertisement obtains certification status (activity 4).

As already mentioned, an important advantage of the SXM testing method which serves as the foundation of our approach is that it 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. This is an important criterion for entrusting the process of verification and certification to the registry. The automated test generation is supported by the JSXM suite of tools (Dranidis 2009).

An additional advantage in our approach that relates to our technological framework is the availability of an open source and standards-based Web service registry (Kourtesis and Paraskakis 2008b) which can be extended with capabilities for functional testing and behavioural verification.

5.3 Validation and service selection

The next activity in the process is for the service requestor to formulate a discovery query and submit it to the service registry (activity 5). The registry will perform some form of matchmaking based on the available advertisements and the specified request, and return the results (activity 6). The discovery and matchmaking method by which the candidate services will be derived is independent from the rest of the approach, and can be based on any existing method. However, a semantically-enhanced service matchmaking method such as the one described in (Kourtesis and Paraskakis 2008) would be strongly encouraged, since it is free of ambiguity, takes more information into consideration, and has the potential of resulting in more accurate matches. In any case, if the registry returns more than one certified services as matching candidates, the requestor must go through a service selection process (activity 7).

As already discussed, the SXM specification that is associated with each of the certified candidate services can be used not only for registry-side verification, but also for requestor-side validation during service selection. A method that enables behavioural validation is model animation through appropriate tools. During animation the requestor feeds the SXM model with sample inputs while observing the current state, transitions, processing functions, memory values, as well as outputs. The sample inputs to be provided for driving the animator can be the actual test data that were generated and used by the service registry at the phase of verification. This would relieve the service requestor from the burden of re-generating the data from the SXM specification. The animation process is readily supported by the existing JSXM tools (Dranidis 2009).

6. Conclusions

Reliability is a fundamental prerequisite for cooperation among peers in a collaborative network, especially in the context of industrial collaborative networks where economic benefits are at stake. For reasons of efficiency, it is typically preferable to delegate the task of evaluating the reliability of prospective or existing network members to entities which act as central authorities for coordination and brokerage, such as coordinators in Virtual Organisation Breeding Environments (Camarinha-Matos 2007). This allows network members to be reassured as to the level of trustworthiness of their prospective business partners by a trusted entity that operates in an accountable and transparent manner.

Evaluating reliability can encompass several different perspectives and layers of abstraction, but one of the most crucial aspects that a central authority needs to consider, from a cooperation perspective, is whether the software systems that support the business processes of some collaborator behave as expected. Expectations concerning the way in which a system should function may arise for several reasons, such as conformance to a particular interaction protocol as prescribed by industrial standards, adherence to the terms of a service contract, or compliance to policies and regulations. For organisations that rely on a contemporary service-oriented

computing infrastructure, evaluating reliability calls for checking that the functionality of the provisioned Web services is conformant to their associated behavioural specifications.

Despite the existence of several standards around Web service technologies, a standardised method for creating behavioural specifications of Web services is currently lacking, along with tools for automating the process of behavioural conformance checking through testing. This paper presents a concrete approach that is aimed at overcoming these deficiencies and supporting the operations of network brokers and coordinators by augmenting Web service registries through the utilisation of formal methods for registry-based functional testing and certification of Web services.

Formal engineering methods for modelling system behaviour, verifying specifications and testing implementations are considered to be among the most central contributions to the advancement of collaborative networks (Camarinha-Matos 2005). In this paper we proposed the use of Stream X-machines (SXMs) as a powerful modelling formalism for constructing the behavioural specification of Web services at the provider-side, in order to facilitate registry-side verification at the time of service publication, and requestor-side validation at the time of service selection.

The particular strengths of the presented approach, compared to other works in the literature, can be summarised in the following. Firstly, a significant advantage of Stream X-machines compared to other behavioural modelling and testing formalisms is in their associated complete testing method, which is guaranteed to reveal all inconsistencies among a specification and an implementation under test, and confirm functional equivalence. Secondly, the SXM specification and the generated test sets can be used not only for registry-side verification, but also for requestor-side validation after discovery and during service selection. Thirdly, the proposed approach can be readily supported by a number of existing tools for SXM modelling, test case generation, verification, and validation, as well as an existing open source service registry implementation for performing semantically-enhanced publication and discovery of services.

Objectives for future research include the consolidation of existing techniques, methods and tools into a comprehensive application framework, experimental validation of the overall approach through a wide range of case studies, and development of suitable connecting components and user-friendly interfaces to yield an all-inclusive solution with industrial applicability.

Acknowledgments

This research work was partially supported by FUSION (Business process fusion based on semantically-enabled service-oriented business applications), a research project funded by the European Commission’s 6th Framework Programme for RTD under contract number FP6-IST-2004-170835 (http://www.fusion-strep.eu/).

References

1. Bertolino, A., Frantzen, I., Polini, A. and Tretmans, J., 2006. Audition of Web Services for Testing Conformance to Open Specified Protocols. *Architecting Systems with Trustworthy Components, Springer LNCS 3938*, 1-25.
2. Camarinha-Matos, L.M. and Afsarmanesh, H., 2005. Collaborative Networks: a New Scientific Discipline. *Journal of Intelligent Manufacturing*, 16, 439-452.

3. Camarinha-Matos, L.M., 2007. Collaborative Networks in Industry: Trends and Foundations. In: P.F. Cunh*a* and P.G. Maropoulos, eds. *Digital Enterprise Technology: Perspectives and Future Challenges.* New York: Springer, 45-56.

4. Chow, T.S., 1978. Testing Software Design Modelled by Finite State Machines. *IEEE Transactions on Software Engineering*, 4, 178-187.

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

6. Dranidis, D., Kourtesis, D. and Ramollari, E., 2007. Formal Verification of Web Service Behavioural Conformance through Testing. *Annals of Mathematics, Computing & Teleinformatics (AMCT)*, 1 (5), 36-43.

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

8. Eilenberg, S., 1974. *Automata, Languages and Machines, Volume A.* New York: Academic Press.

9. Farrell, J. and Lausen, H. (eds), 2007. Semantic Annotations for WSDL and XML Schema (SAWSDL). W3C Recommendation.

10. Heckel, R. and Mariani, L., 2005. Automatic Conformance Testing of Web Services. *Fundamental Approaches to Software Engineering, Springer LNCS 3442*, 34-48.

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

12. Ipate, F. and Holcombe, M., 1997. An integration testing method that is proved to find all faults. *International Journal of Computer Mathematics*, 63, 159-178.

13. Kapeti, E. and Kefalas, P., 2000. A Design Language and Tool for X-Machine Specification. Advances in Informatics, *Proceedings of the 7th Hellenic Conference on Informatics (HCI '99)*, 134-145.

14. Kefalas, P., 2000. X-Machine Definition Language: User Manual, version 1.6. *Technical Report WP-CS07-00*, CITY College.

15. Keum, C., Kang, S. and Ko, I.Y., 2006. Generating Test Cases for Web Services using Extended Finite State Machine. *Testing of Communicating Systems, Springer LNCS 3964*, 103-117.

16. Kourtesis, D. and Paraskakis, I., 2008. Web Service Discovery in the FUSION Semantic Registry. *Business Information Systems 2008, Springer LNBIP 7*, 285–296.

17. Kourtesis, D. and Paraskakis, I., 2008. Combining SAWSDL, OWL-DL and UDDI for Semantically Enhanced Web Service Discovery. *The Semantic Web: Research and Applications, Springer LNCS 5021*, 614–628.

18. Kourtesis, D., Ramollari, E., Dranidis, D. and Paraskakis, I., 2008. Discovery and Selection of Certified Web Services through Registry-Based Testing and Verification. *In:* L. Camarinha-Matos and W. Pickard, eds. *Pervasive Collaborative Networks*, Boston: Springer, 473-482.

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

20. Msanjila, S.S., Afsarmanesh, H., 2007. Modelling Trust Relationships in Collaborative Networked Organisations. *International Journal of Technology Transfer and Commercialisation*, 6 (1), 40-55.

21. Msanjila, S.S., Afsarmanesh, H., 2007. HICI: An Approach for identifying Trust Elements - The Case of Technological Perspective in VBEs. *Proceedings of the 2nd International Conference on Availability, Reliability and Security (ARES 2007)*, 757-764.

22. Ramollari, E., Kourtesis, D., Dranidis, D. and Simons, A.J.H., 2009. Leveraging Semantic Web Service Descriptions for Validation by Automated Functional Testing. The Semantic Web: Research and Applications, Springer LNCS 5554, 593-607.

23. Sinha, A. and Paradkar, A., 2006. Model-based Functional Conformance Testing of Web Services Operating on Persistent Data. Proceedings of Workshop on Testing, Analysis and Verification of Web Services and Applications (TAV-WEB'06), 17-22.

24. Tsai, W.T., Paul, R., Cao, Z., Yu, L., Saimi, A. and Xiao, B., 2003. Verification of Web Services using an Enhanced UDDI Server. *Proceedings of 8th IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS 2003)*, 131-138.

Appendix

In this appendix we provide the full specification of the Stream X-machine model for the example Web service that was presented in this paper, using XMDL-O notation.

Inputs

```
#event loginRequest(usr:string,pw:string).
#event logoutRequest().
#event createOrderRequest().
#event cancelOrderRequest().
#event
addItemRequest(itemId:string,qty:natural0).
#event
removeItemRequest(itemId:string,qty:natural0
).
#event getQuoteRequest().
#event confirmOrderRequest().
#event rejectOrderRequest().
```

Outputs

```
#output (messages).

#type               messages              =
{loginOk,loginFailed,loggedOut,orderCreated,
orderCanceled,itemAdded,itemQtyIncreased,ite
mRemoved,removeFailed,itemQtyDecreased,lastI
temRemoved,quoteEmpty,quotePartial,quoteFulf
illed,orderRejected,orderConfirmed}.
```

States

```
#states    =    {initial,    authenticated,
empty_order,   filling_order,   pending_conf,
order_placed}.

#init_state {initial}.
```

Transition Function

```
#transition (initial,loginFailed)=initial.
#transition (initial,loginOK)=authenticated.
#transition (authenticated,logout)=initial.
#transition
(authenticated,createOrder)=empty_order.
#transition
(empty_order,cancelOrder)=authenticated.
#transition
(empty_order,addOrderLine)=filling_order.
#transition
(filling_order,cancelOrder)=authenticated.
#transition
(filling_order,addOrderLine)=filling_order.
#transition
(filling_order,increaseItemQty)=filling_orde
r.
#transition
(filling_order,removeOrderLine)=filling_orde
r.
#transition
(filling_order,decreaseItemQty)=filling_orde
r.
#transition
(filling_order,removeError)=filling_order.
#transition
(filling_order,removeLastOrderLine)=empty_or
der.
#transition
(filling_order,getQuoteEmpty)=filling_order.
#transition
(filling_order,getQuotePartial)=pending_conf
.
#transition
(filling_order,getQuoteFulfilled)=pending_co
nf.
#transition
(pending_conf,rejectOrder)=filling_order.
#transition
(pending_conf,confirmOrder)=order_placed.
```

Memory

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

#class InventoryItem {
  id: string,
  qty_in_stock: natural0,
  qty_on_hold: natural0,
}.

#class OrderItem {
  id: string,
  qty_requested: natural0,
  qty_reserved: natural0,
}.

#objects:
  account1: Account,
  accounts: set_of Account,
  inventoryItem1: InventoryItem,
  inventoryItem2: InventoryItem,
  inventory: set_of InventoryItem,
  order: set_of OrderItem.

#init_values:
  account1.username <- "usr1",
  account1.password <- "pwd1",
  inventoryItem1.id <- 1001,
  inventoryItem1.qty_in_stock <- 100,
  inventoryItem1.qty_on_hold <- 0,
  inventoryItem2.id <- 1002,
  inventoryItem2.qty_in_stock <- 50,
  inventoryItem2.qty_on_hold <- 0,
  accounts <- {account1},
  inventory       <-       {inventoryItem1,
  inventoryItem2},
  order <- emptySet.
```

Processing Functions

```
#fun loginOK( loginRequest(?usr,?pw) )=
  if       not_isempty       ?found       and
?current_user.pw=?pw
  then
   (loginOk)
  where
   ?found  <-  select  (usr=?usr,  accounts)
and
   ?current_user <- head(?found).


#fun loginFailed( loginRequest(?usr,?pw) )=
  if isempty ?found
```

```
  then
   (loginFailed)
  where
   ?found <- select (usr=?usr, accounts).

#fun logout( logoutRequest() )=
  (loggedOut).


#fun createOrder( createOrderRequest() )=
  (orderCreated).


#fun cancelOrder( cancelOrderRequest() )=
  (orderCanceled).


#fun                            addOrderLine(
addItemRequest(?itemId,?qty) )=
  if isempty ?found
  then
   (itemAdded)
  update
   order  <-  new  OrderItem(?itemId,?qty,0)
addsetelement order
  where
   ?found <- select (id=?itemId, order).


#fun                          increaseItemQty(
addItemRequest(?itemId,?qty) )=
  if not_isempty ?found
  then
   (itemQtyIncreased)
  update
   ?orderline.qty_requested              <-
?orderline.qty_requested + ?qty
  where
   ?found <- select (id=?itemId, order) and
   ?orderline <- head(?found).


#fun                          removeOrderLine(
removeItemRequest(?itemId,?qty) )=
  if       not_isempty       ?found       and
?orderline.qty_requested ≤ ?qty
  then
   (itemRemoved)
  update
   order <- ?orderline delsetelement order
  where
   ?found <- select (id=?itemId,order) and
   ?orderline <- head(?found).


#fun                          decreaseItemQty(
removeItemRequest(?itemId,?qty) )=
  if       not_isempty       ?found       and
?orderline.qty_requested > ?qty
  then
   (itemQtyDecreased)
  update
   ?orderline.qty_requested              <-
?orderline.qty_requested - ?qty
  where
   ?found <- select (id=?itemId, order) and
   ?orderline <- head(?found).


#fun                            removeError(
removeItemRequest(?itemId,?qty) )=
  if isempty ?found
  then
   (removeFailed)
  where
   ?found <- select (id=?itemId, order).


#fun                    removeLastOrderLine(
removeItemRequest(?itemId,?qty) )=
  if       not_isempty       ?found       and
?orderline.qty_requested ≤ ?qty and ?count =
1
  then
   (lastItemRemoved)
  update
   order <- ?orderline delsetelement order
  where
   ?found <- select (id=?itemId,order) and
   ?orderline <- head(?found) and
   ?count <- cardinality order.


#fun getQuoteEmpty( getQuoteRequest() )=
  if     conjunction     (fn     ?id     =>
(?invline.qty_in_stock-
?invline.qty_on_hold=0)
   where
   ?invline      <-      head(select(id=?id,
inventory), ?order_ids))
  then
   (quoteEmpty)
  where ?order_ids <- project(id, order).


#fun getQuotePartial( getQuoteRequest() )=
  if
  disjunction       (fn       ?id       =>
(?invline.qty_in_stock-
?invline.qty_on_hold<?line.qty_requested
   where
   ?line <- select(id=?id, order) and
   ?invline  <-  select(id=?id,  inventory)),
?order_ids)
  and
  disjunction       (fn       ?id       =>
(?invline.qty_in_stock-
?invline.qty_on_hold>0)
   where
   ?invline   <-   select(id=?id,   order),
?order_ids)
  then
   (quotePartial)
  update
   inventory <- map ( fn ?invitem -> ( new
InventoryItem(                  ?invitem.id,
?invitem.qty_in_stock,  ?invitem.qty_on_hold
+ ?increase)
   where  ?increase  <-  (if  ?invitem.id
belongs ?order_ids
    then
     minimum    (    ?orditem.qty_requested,
?qty_available )
    else 0) ) ) and
    ?orditem                              <-
head(select(id=?invitem.id,orders)) and
    ?qty_available                        <-
?invitem.qty_in_stock-?invitem.qty_on_hold,
inventory )

  and
   order  <-  map  (  fn  ?orditem  ->  (new
OrderItem(?orditem.id,
?orditem.qty_requested, ?reserved ) ) where
    ?reserved          <-          minimum(
?orditem.qty_requested, ?qty_available ) and
    ?qty_available                        <-
?invitem.qty_in_stock-?invitem.qty_on_hold
and
```

```
    ?invitem <- head(select(id=?orditem.id,
inventory)), order)

  where ?order_ids <- project(id, order).


#fun getQuoteFulfilled( getQuoteRequest() )=
  if     conjunction     (fn     ?id     =>
(?invline.qty_in_stock-
?invline.qty_on_hold≥?line.qty_requested
   where
   ?line <- select(id=?id, order) and
   ?invline  <-  select(id=?id,  inventory)),
?order_ids)
  then
   (quoteFulfilled)
  update
   inventory <- map (fn ?invitem => ( new
InventoryItem(                  ?invitem.id,
?invitem.qty_in_stock,  ?invitem.qty_on_hold
+ ?increase)
   where  ?increase  <-  (if  ?invitem.id
belongs ?order_ids
    then
     ?orditem.qty_requested else 0) )
  and
    ?orditem                              <-
head(select(id=?invitem.id,orders)),
inventory)

  and

   order  <-  map  (fn  orditem  =>  (new
OrderItem(?orditem.id,
?orditem.qty_requested,
?orditem.qty_requested)), order)
    where
    ?order_ids <- project(id, order).
```

```
#fun rejectOrder( rejectOrderRequest() )=
  (orderRejected)

update

  inventory <- map ( fn ?invitem -> ( new
InventoryItem(                  ?invitem.id,
?invitem.qty_in_stock,  ?invitem.qty_on_hold
- ?decrease)
   where  ?decrease  <-  (if  ?invitem.id
belongs           ?order_ids           then
?orditem.qty_reserved else 0) ) ) and
   ?orditem                              <-
head(select(id=?invitem.id,orders)),
inventory )

  and

  order  <-  map  (  fn  ?orditem  ->  (new
OrderItem(?orditem.id,
?orditem.qty_requested, 0 ) ), order).


#fun confirmOrder( confirmOrderRequest() )=
  (orderConfirmed)

  update

   inventory <- map ( fn ?invitem -> ( new
InventoryItem(                  ?invitem.id,
?invitem.qty_in_stock      -      ?decrease,
?invitem.qty_on_hold)
    where  ?decrease  <-  (if  ?invitem.id
belongs           ?order_ids           then
?orditem.qty_reserved else 0) ) ) and
    ?orditem                              <-
head(select(id=?invitem.id,orders)),
inventory ).
```