Towards Model-Based Testing of Web Services

Lars Frantzen [1,3]

Instituto di Scienza e Tecnologie della Informazione "Alessandro Faedo"
Consiglio Nazionale delle Ricerche, Pisa – Italy

Jan Tretmans [4] René de Vries [2,5]

Institute for Computing and Information Sciences
Radboud University Nijmegen – The Netherlands

Abstract

Complex interactions between Web Services involve coordinated sequences of operations. Clients of the provided services must be aware of the underlying *coordination protocol* to smoothly participate in such a coordinated setup. In this paper we discuss on a running example how such protocols may also serve as the input for Model-Based Testing of Web Services. We propose to use Symbolic Transition Systems and the rich underlying testing theory to approach modelling and testing the coordination. We further indicate where theoretical and technical gaps exist and point to several research issues.

Key words: Web Service, Coordination Protocol, Model-Based Testing, Symbolic Transition System

1 Introduction

A Web Service (WS) never walks alone: it publishes interfaces to potential users of the provided service. To be utilised efficiently these interfaces need to be described in a way accessible to machines, i.e. in a language with formal syntax and semantics. In this paper we want to show how such interface

[1] Lars Frantzen is supported by the Marie Curie Network TAROT (MRTN-CT-2004-505121) and by the Netherlands Organisation for Scientific Research (NWO) under project STRESS.
[2] René de Vries carried out this work as part of the TANGRAM project under the responsibility of the Embedded Systems Institute. Tangram is partially supported by the Netherlands Ministry of Economic Affairs under grant TSIT2026.
[3] Email: `lars.frantzen@isti.cnr.it`
[4] Email: `tretmans@cs.ru.nl`
[5] Email: `rgv@cs.ru.nl`

specifications can be exploited to serve as the input for *Model-Based Testing* (MBT).

With testing we mean the act of performing experiments with the WS, aiming at either finding faults, or gaining adequate confidence in the conformance of the WS to its interface specification. Model-based testing is a kind of black-box testing, where these experiments are automatically generated from the formally described interface specification, and subsequently also automatically executed. By founding our testing approach on well established formal testing theories, we have the advantage of automating both test generation and test execution, and moreover, it is precisely defined what exactly we mean with conformance.

One crucial issue when following this path of model-based testing is the unavoidable trade-off between increasing *testability* and *information hiding*. By just publishing very general information about the provided service, e.g. mere signatures of the interface operations, one can completely hide the possibly proprietary business logic which constitutes the operations' implementation. Furthermore, the implementation can change freely as long as it still continues sticking to the interface description. Such kind of interface specifications can be expressed in the Web Service Definition Language (WSDL) [6]. When it comes to testing, the only information we can exploit in such a setting is the WSDL-file; we have no knowledge about any internal structure (e.g. the source code), nor about any ordering among the different operations. When testing in this setting, we can only test each provided operation separately, applying classical black-box testing techniques, such as equivalence partitioning or boundary value analysis.

This separate testing of each provided operation is fine as long as the operations the WS offers are independent from each other, i.e. all operations can be called at any time, without influencing each others behaviour. For instance, think of a WS translating Italian and Spanish words into English, thus offering two independent query operations. Testing such a WS boils down to testing each operation separately.

In many non-trivial WSs the invocations of operations are not independent; there are restrictions on the dynamics of operations, i.e., the invocations have to obey some ordering. We will use the term *WS protocol* to denote the specification of the legal orderings of invocations of a WS. Continuing the example above, the dictionary WS could be restricted to registered users. Thus, before being able to (successfully) invoke the query operations, a user first has to log into the system via a dedicated operation.

A mere collection of operation-signatures given in the WSDL only describes the static aspect of invocations; it is not sufficient to specify the allowed sequences of invocations. Thus, it is crucial that the WS offers an additional source of information, which can be accessed to discover the WS protocol a user has to obey. To make clients aware of the protocol, its description should be available in WS registries.

In order to proceed towards our goal of automatic, model-based testing of web services, WS protocols must be formally modelled. One commonly used model for specifying WS protocols is a state machine. We will propose to use a special variant of a state machine, namely a *Symbolic Transition System* (STS) as introduced in [7]. An STS has states and labelled transitions between states modelling the actions, i.e. the inputs and outputs, of the system. Both states and actions can be parameterised with data variables, and predicates on these variables may guard the transitions.

The use of STSs for specifications allows to exploit the well established STS-based testing theory and algorithms of [7]. These include a precise definition of conformance of a WS implementation with respect to its specification using the *implementation relation* **ioco** [13], an algorithm for the generation of test cases from an STS specification, and notions of soundness and exhaustiveness of the generated test cases. Moreover, several testing tools nowadays implement this test generation algorithm it, e.g. TorX [2] and the TGV-based tools [11]. The tool TorX generates and executes test cases on-the-fly, which means that instead of firstly computing a set of test cases from the STS, and then applying them to the System Under Test (SUT), it generates a single test event in each step, and immediately executes it to the SUT. As a consequence the state space explosion when generating test cases, is avoided, see also [2].

The STS model as defined in [7] turns out to be rich enough to formally model a WS protocol between a single client and web service; this will be elaborated in Sect. 3. More useful web services are commonly built by coordinating many simpler web services. This implies that we must take into account a WS protocol comprising several WSs which interact with each other. In Sect. 4 we will analyse the shortcomings of the currently defined STS model and its corresponding testing theory for testing these multi-WS interactions, and we will point to some first approaches to deal with these cases.

The goal of the paper is to discuss how current model-based testing techniques might be applied to testing web services, and to indicate where theoretical and technical gaps exist. It tries to motivate a promising path and its cornerstones, hopefully leading to a complete and formal treatment of testing web services in the future.

Overview

This paper is structured as follows. Section 2 introduces the running example used throughout the paper. In Sect. 3 we discuss the STS-model, and exemplify how to use it for modelling and testing the conversations between a single client and a WS. Section 4 shows further approaches to deal with a setting comprising several WSs. Section 5 concludes the paper.

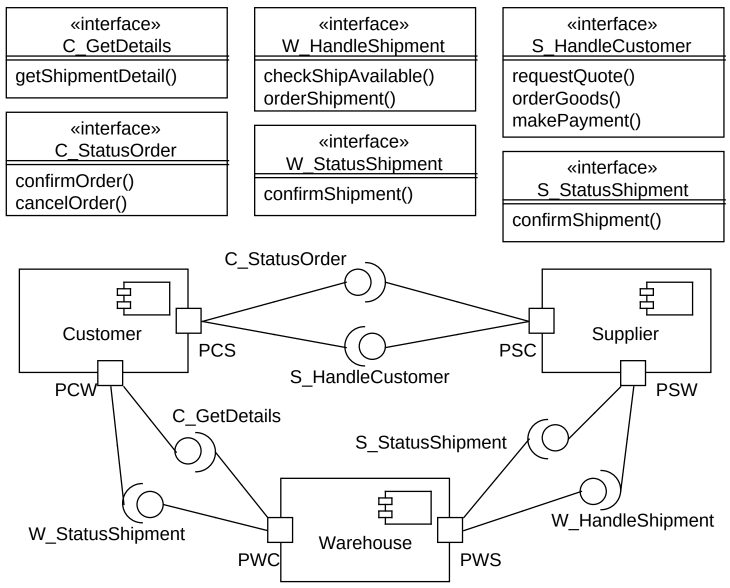

Fig. 1. The procurement protocol setup

2 Case: Procurement Scenario

In this section we briefly review the basics of WSs which are relevant for our MBT approach. For testing it is required to identify the Points of Control and Observation (PCOs), i.e. points where interaction occurs. Secondly, we need to identify the direction of interaction, i.e. whether there is an input (stimulus) or an output (observation). To analyse this, we model a WS in the realm of UML components, its ports and interface connectors. We review WSs by considering a running example of the Procurement Scenario adopted from [1].

Consider a customer who needs to order goods from a supplier. The customer orders at the supplier, which processes the order. Then the supplier requests a warehouse to deliver the goods to the customer. Now assume that we automate this scenario in a Business to Business setting, where every party has its own WS to interact with its peers. This is visualised in Figure 1 as an UML 2.0 Component Diagram.

We find three WSs `Customer`, `Supplier` and `Warehouse`. The points where we can connect to and interact with a WS are called *ports* (in Figure 1: `PCS`, `PCW`, `PWC`, `PWS`, `PSC` and `PSW`).

At a port we can execute *operations* by means of interaction. The direction of communication of these operations can be differently. We can distinguish among *one-way*, *notification*, *request-response*, and *solicit-response*. One-way can be considered as being able to accept a single input message from a peer WS. Notification can be seen as doing a single output message by the WS. Request-response corresponds to an API like invocation, i.e. the operation consists out of first accepting an input and then returning a result to the

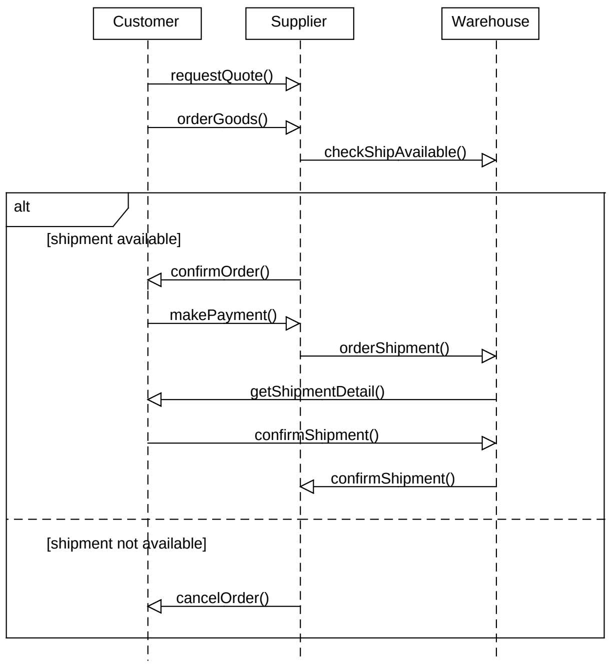

Fig. 2. The procurement protocol

caller. The solicit-response operation can be considered as the WS taking initiative to make an invocation to another WS, i.e. make a response (output) and waiting for an answer (input). Synchronous interactions are modelled via *one-way* and *notification* operations; asynchronous interactions correspond to *request-response* and *solicit-response* operations. In this paper we group operations at a particular port into *interfaces*.

Every port consists of several interfaces, e.g. `PCW` consists out of the interfaces `C_GetDetails` and `W_StatusShipment`. For now we consider an interface that takes initiative to communicate an *output interface* (i.e. solicit-response and notification operations), while an *input interface* is accepting messages (i.e. request-response and one-way operations). For the given Procurement Scenario we assume all operations to be synchronous, i.e. we only deal with solicit-response and request-response operations. For the `Customer` WS, `C_GetDetails` and `W_StatusShipment` are respectively an input and output interface. Typically the signature of the operations are described by WSDL. In our example the operation-signatures at the `W_HandleShipment` interface are `checkShipAvailable()` and `orderShipment()`.

The behaviour of a use case of the Procurement Scenario can be described by an UML 2.0 sequence diagram as is done in Figure 2. The `Customer` WS can connect to the `Supplier` WS to inform about the price, availability, delivery dates etc. of goods by means of the `requestQuote()` operation. Then it places the order by a `orderGoods()` operation. The `Supplier` WS checks at the `Warehouse` WS whether there is a shipment available by a `checkShipAvailable()` operation. Based on the result the `Supplier` WS responses the `Customer` WS a `cancelOrder()` operation (e.g. he ran out of stock) or confirms the order by a `confirmOrder()` operation. If the supplier can deliver the following sequence occurs: the customer pays (`makePayment()`), the shipment is ordered to the warehouse (`orderShipment()`), the shipment details are provided to the customer (`getShipmentDetail()`), the customer confirms its order (`confirmShipment()`) and the warehouse confirms the shipment to the supplier (`confirmShipment()`).

3 Testing Conversations Between a Client and a WS

In this section we exemplify how to use STSs for modelling and testing the dynamics of conversations between a single client and a WS.

Coming to WS protocols, keeping the information hiding principle is not as straightforward as it is with mere operation signatures. The information revealed to the user depends on the chosen specification model, and on the part of the WS setup which is chosen as constituting the SUT. The more complex this part is, the more meaningful the cooperation of WSs can be tested. In the next subsections we start with very simple models. The first considers only a single input interface, where each operation invocation is modelled by an action. Then, in Sect. 3.3, we consider ports, but we then still model each invocation by a single action in the STS. Finally, starting in Sect. 3.5, we discuss testing where data is added to invocations, and each invocation is split into two actions: one to model the call or start of the invocation, and one to model the end or return of the invocation (also called *request* or *solicit*, and *response*, respectively.

3.1 Symbolic Transition Systems

Our specification models of choice are STSs. Such an STS is a state machine incorporating an explicit notion of data and data-dependent control flow (such as guarded transitions), founded on first order logic. The STS model clearly reflects the *Labelled Transition System* (LTS) model, which is done to smoothly transfer LTS-based test theory concepts to an STS-based test theory. The underlying first order structure gives formal means to define both the data part algebraically, and the control flow part logically. This makes STSs a very general and potent model for describing several aspects of reactive systems. We do not give here a formal definition of the syntax and semantics of STSs due to its extent, but motivate their usage in the following examples. For a formal definition we refer to [7].

3.2 Testing a Single Input Interface

Seeing a WS as being an isolated component, possibly neglecting the role it may play in a coordinated set up, a WS protocol may only refer to operations from the WSs input interface. Given a simple case where the conversation with the user does not happen via output interfaces, this might be a reasonable thing to do. Our example already shows that in practice for many non trivial examples a WS cannot be specified and tested from such an isolated point of view.

Take for example the input interface `S_HandleCustomer` from the `Supplier` WS. Suppose one wishes to test the conversations between a `Supplier` and a `Customer` based on this interface. A WS protocol would specify that at first a customer has to invoke the `requestQuote` method, followed by an `orderGoods` method. From this point in time, one cannot express anymore if the client is supposed to initiate a `makePayment` call because this depends on the answer from the `Supplier` as being given via the output interface `C_StatusOrder` (which is an input interface for the customer). We have to observe if the supplier sends the message `confirmOrder` or `cancelOrder` to the customer to make a judgement here. Thus, a meaningful specification covering more than a prefix of a conversation cannot be given without considering at least the output interface `C_StatusOrder`.

Thence, a specification only comprising input interfaces does strictly uphold the information hiding principle, but does in general not constitute an entity one can utilise for MBT.

3.3 Specifying a Single Port

We will focus here on specifying the `PSC` port. This bi-directional port comprises the input interface `S_HandleCustomer`, and `C_StatusOrder`, the output interface. In this setting the tester plays the role of the `Customer` testing the `Supplier`. Thus, we can use the operations given in these two interfaces. Now we can express the protocol we were not able to give when restricting to just the input interface of the `Supplier` WS.

In Fig. 3 you find a first STS specification of the `PSC` port focussing only on specifying the legal ordering of the operations. In so doing we can neglect their return values since we are not interested in the data communicated. Thus, every operation `in` from the input interface is mapped to the input action `?in`, and every operation `out` from the output interface is mapped to the output action `!out`. In this manner we can give the shown state machine and define the legal ordering of the operations. Note that there is one special label called `(internal trigger)`. This label corresponds to an internal action of the `Supplier` which is not observable in our current setting since it comprises only the `Customer` and the `Supplier`. It refers to the choice of either sending a `!cancelOrder` or a `!confirmOrder` to the `Customer`. Remembering the procurement protocol as being given in Fig. 2, this choice depends on the response from the `Warehouse` if the shipment is available or not. This part of the protocol is out of the scope in our current setting, hence we have to model this choice as a nondeterministic one. Such an `(internal trigger)` usually

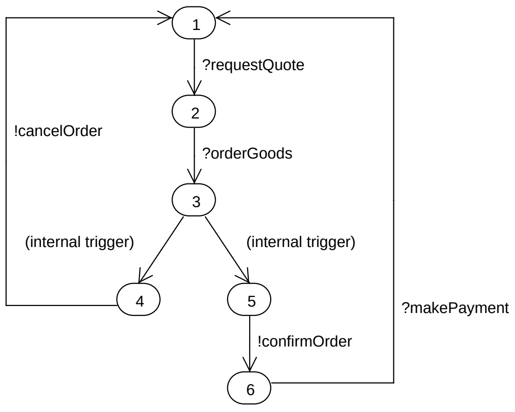

Fig. 3. A basic STS specification for the `PSC` port

is referred to as an τ-action in the underlying theories.

3.4 Testing a Single Port

The given STS is a sheer specification of the order of the interface operations. The business logic revealed via such a specification of legal transitions is more than marginally. It is mandatory to at least inform potential user about the intended orders of operation calls to allow for a smooth communication. Since the data exchanged is completely omitted, the user has no access to knowledge like how it is determined by the `Supplier` if a shipment is cancelled or confirmed, etc.

The value of such a specification for testing a port is very limited. Since no information is given about the data exchanged via the operations, a test system has no access to any kind of indicators on how it should assemble or verify messages sent to, or received from the `Supplier`. Thus, such kind of specifications are more related to monitoring the conversations between WSs on the abstraction level of operation calls, auditing only the calls without looking further into the data exchanged. This can still be of high value to detect elementary communication faults in a coordinated setup.

3.5 Specifying a Single Port Comprising Data

In the basic STS example above we have not used any of the symbolic features of an STS. In fact, we could have modelled the same state machine as a simple LTS. To really test the specific details of a system we have to take into account the data exchanged. In the realm of distributed systems we have the complex situation of several ways and kinds which can be exploited to exchange data. For instance there can be synchronous and asynchronous message types. As we have seen an STS has in principle an asynchronous nature since every transition realises either an input action or an output action. This is fine since synchronous communication involves a tight integration and dependency of the interacting WSs, which is not acceptable in most industrial-strength settings. Still there might be cases where a synchronous setting is favoured, hence the applied modelling and testing techniques should support both kinds of interaction.

In the realm of STSs it is straightforward to model a synchronous operation as an input action representing the method call, followed by an output action representing the returned value (which might of type `void`, i.e. a mere return of control). For instance take the `requestQuote` operation from the input interface. To model the data flow we need to know the complete signature of this operation, including types. The `requestQuote` operation gets an object of type `QuoteRequest` as input, and returns an object of type `Quote`. The complete signature is then `requestQuote(r:quoteRequest):Quote`. To model this operation in an STS we map it to the typed input action `?requestQuote <r:quoteRequest>`, and typed output action `!requestQuote<q:Quote>`. The inverse case shows up when the tested WS calls a method from the output interface. Here, the call is modelled as an output action, and the return is modelled as an input action. While in this two-party setting the splitting of a method call into two actions may appear as an redundant effort, we will see in the next section that the asynchronous nature of STSs is of utmost value when dealing with more then just two parties.

In Fig. 4 you find an STS specification of the `PSC` port specifying much more than just the legal ordering of the transitions. The data signatures of the corresponding interfaces are also given. Each STS transition consists of three parts: first the name of an input- or output action together with its parameters; next a guard talking about the parameters and the internal variables; and finally an update of the internal variables.

Such internal variables are a crucial concept for having a natural and powerful specification model. These concepts sometimes cause confusion when the strict duality between specification models and implementation models is overlooked. Of course, a black box specification must not refer to the real implementation details like variables which really exists in the implementation. Specification variables like internal STS variables are used to abstractly model the state of the SUT and have not any kind of semantical relation to real variables from the SUT.

We exemplify the STS concept on the operation `requestQuote`. The invocation of the method corresponds to the transition from state 1 to state 2. First, the label mentions the input action `?requestQuote<r:QuoteRequest>`. Here we refer with `r` to the input value of type `QuoteRequest`. Next, the guard `[r.quantity>0]` constrains the attribute `quantity` of `r` to be a positive integer. The variable `r` is a special kind of variable, called interaction variable, which is local to an action, not global to the whole STS as internal variables are. Hence we have to save the value communicated via `r` in an internal variable `quoteRequested`. This is done via the assignment `quoteRequested:=r`.

The succeeding transition from state 2 to state 3 corresponds to the re-

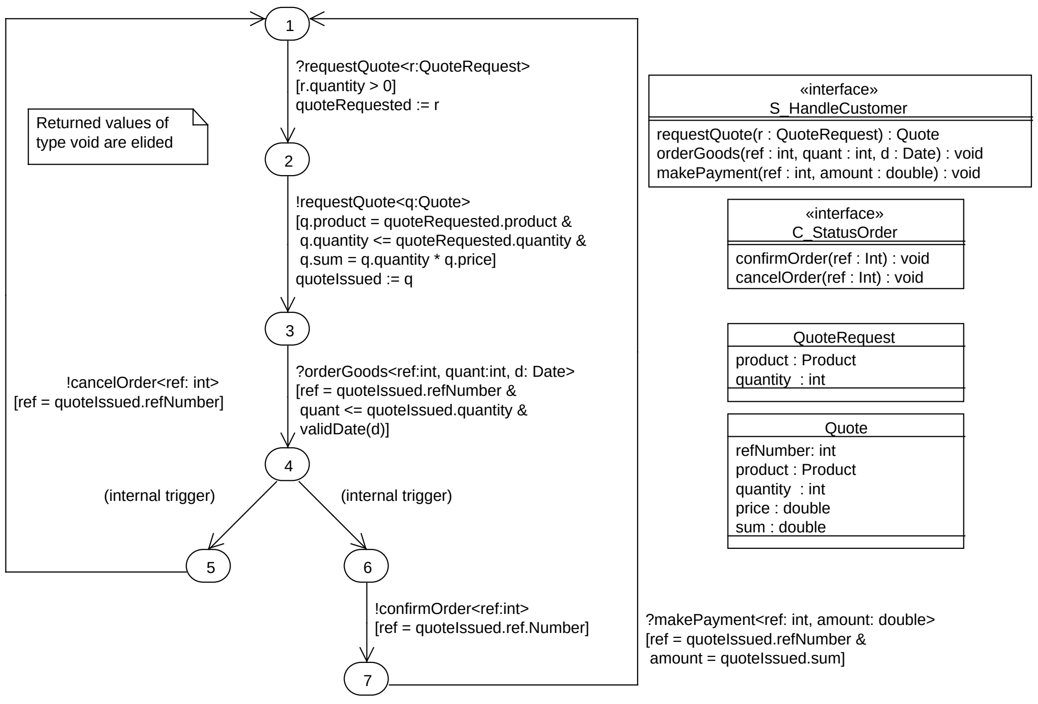
1
?requestQuote<r:QuoteRequest>
[r.quantity > 0]
quoteRequested := r
Returned values of type void are elided
2
!requestQuote<q:Quote>
[q.product = quoteRequested.product &
q.quantity <= quoteRequested.quantity &
q.sum = q.quantity * q.price]
quoteIssued := q
3
!cancelOrder<ref: int>
[ref = quoteIssued.refNumber]
?orderGoods<ref:int, quant:int, d: Date>
[ref = quoteIssued.refNumber &
quant <= quoteIssued.quantity &
validDate(d)]
4
(internal trigger)
(internal trigger)
5
6
!confirmOrder<ref:int>
[ref = quoteIssued.ref.Number]
7
?makePayment<ref: int, amount: double>
[ref = quoteIssued.refNumber &
amount = quoteIssued.sum]
«interface»
S_HandleCustomer
requestQuote(r : QuoteRequest) : Quote
orderGoods(ref : int, quant : int, d : Date) : void
makePayment(ref : int, amount : double) : void
«interface»
C_StatusOrder
confirmOrder(ref : Int) : void
cancelOrder(ref : Int) : void
QuoteRequest
product : Product
quantity : int
Quote
refNumber: int
product : Product
quantity : int
price : double
sum : double

Fig. 4. A detailed STS specification for the `PSC` port

turned value of the `requestQuote` operation, which is an object of type `Quote`, referenced by the interaction variable `q`. The guard here ensures that returned quote deals with the same product as mentioned in the requested quote. It further constrains the offered quantity to be less or equal to the requested one, and ensures that the mentioned sum of the quote equals the quantity times price per item. This is done by relating the internal variable `quoteRequested` and the interaction variable `q`. Finally, the offered quote is saved in the internal variable `quoteIssued`.

We have to reconsider here the principle of information hiding. The presented STS clearly presents a more detailed view on the specified system. This is mainly due to the guards which reveal constraints we can assume when dealing with the system. These constraints are crucial for testing the system, but may not be crucial for mere coordination needs. It is up to the specific situation how detailed a specification is supposed to be, depending on which aim it serves.

3.6 Testing a Single Port Comprising Data

Given a specification of a port in the STS formalism, we can apply the test generation algorithm as given in [7]. The first action of a tester implementing that algorithm would be to request a quote from the `Supplier` for an arbitrary product with a positive quantity. Next, the tester will check the returned `Quote` object and verify that the guard of the concerned transition from state

```
STS is in stable state - Choosen to give an input
Giving this testevent to the STS:
 02/05/06 18:47:33:650: ?requestQuote[QuoteRequest with quantity 551]
STS Manager: 1 instantiated location(s):
 Location: 2: [QuoteRequest with quantity 551, Quote with quantity 0]
Applying input to SUT.
Will apply output response from SUT to STS.
Giving this testevent to the STS:
 02/05/06 18:47:35:343: !requestQuote[Quote with quantity 505]
STS Manager: 1 instantiated location(s):
 Location: 3: [QuoteRequest with quantity 551, Quote with quantity 505]
STS is in stable state - Decided to wait for quiescence.
STS Manager: Found Quiescence.
Giving this testevent to the STS:
 02/05/06 18:47:35:765: (quiescence)
STS Manager: 1 instantiated location(s):
 Location: 3: [QuoteRequest with quantity 551, Quote with quantity 505]
STS is in stable state - Choosen to give an input
Giving this testevent to the STS:
 02/05/06 18:47:35:949: ?orderGoods[1, 438]
STS Manager: 1 instantiated location(s):
 Location: 4: [QuoteRequest with quantity 551, Quote with quantity 505]
Applying input to SUT.
Will apply output response from SUT to STS.
Giving this testevent to the STS:
 02/05/06 18:47:36:124: !orderGoods[]
STS Manager: 2 instantiated location(s):
 Location: 5: [QuoteRequest with quantity 551, Quote with quantity 505]
 Location: 6: [QuoteRequest with quantity 551, Quote with quantity 505]
STS Manager: Output found.
Giving this testevent to the STS:
 02/05/06 18:47:36:125: !cancelOrder[1]
...
```

Fig. 5. Testing the PSC port on-the-fly

2 to 3 is fulfilled. If this is not the case, the test stops with verdict `Fail`. For instance, this is the case when the quote names a quantity greater than the requested one. Given a valid return the tester sends next a suited `?orderGoods` message. This message returns the control without communicating data, these `void`-returns are elided in the STS.

Next, the tester does not know if the SUT is in state 5 or 6 due to a nondeterministic internal trigger. Hence, both receiving a `!cancelOrder` and a `!confirmOrder` message is conformant to the specification. In practice the testing continues in this manner until either a fault is discovered via verdict `Fail`, or the testing is stopped after a predefined halting criteria. Figure 5 shows a simplified extract of the output of a prototype tool implementing the exemplified test generation algorithm.

For the conformance testing in general it remains to be evaluated how such halting criteria should be defined. It will also depend on the given application domain and its inherent security demands which specific halting criteria is considered sufficient. There are several well known halting criteria for model-based testing, mainly concerning coverage of the specification ingredients (states, transitions, evaluation criteria for the guards, etc.). Also more specific testing scenarios (called test purposes) might be of high value.

4 Research Issues

In the previous section we have addressed the basics of testing WSs using MBT techniques. Mostly these techniques have been developed for testing reactive systems. In this section we discuss adaptations and extensions of the underlying theory to improve our approach for testing WSs in the full setup, consisting of multiple ports, multiple WSs, etc. Here we are confronted with the big picture as being given in Fig. 1. To keep the specifications simple we will focus again on basic STS models which only show the legal invocations of operations. Every argument presented here is equally valid for STSs comprising data.

4.1 Conformance Between Connected Ports

Following the running example we have focussed up to now on the `PSC` port of the `Supplier` WS. One obvious observation here is, that every port communicates with a partner port of another WS. For instance the `PSC` port talks with the `PCS` port of the `Customer`. Such partner ports are symmetric in the sense that both share the same set of interfaces, but what is an input interface for one is an output interface for the other. The `PSC` and `PCS` ports share the `C_StatusOrder` and `S_HandleCustomer` interfaces, in which `C_StatusOrder` is an input interface for the `Customer` and an output interface for the `Supplier`. The `S_HandleCustomer` is an output interface for the `Customer` and an input interface for the `Supplier`.

We may take advantage of this symmetry to define a notion of conformance between connected ports. Since connected ports have symmetric interfaces their underlying STS specifications also have symmetric actions. What is an input action for one is an output action for the other. Hence, swapping the set of input and output actions in such an STS yields a specification of its partner port. For instance one can swap inputs and outputs in the `PSC` specification of Fig. 4, i.e. visually all interrogation marks preceding an action become an exclamation mark, and vice versa. This resulting STS now constitutes a specification of the partner `PCS` port. So doing may allow us to formally define conformance of connected ports based on a formal conformance relation like **ioco**.

4.2 Integration of Service Specifications

A WS might consists out of several ports. Thus, if we want to test not only a single port, but a WS as a whole, we need to have specifications of all ports

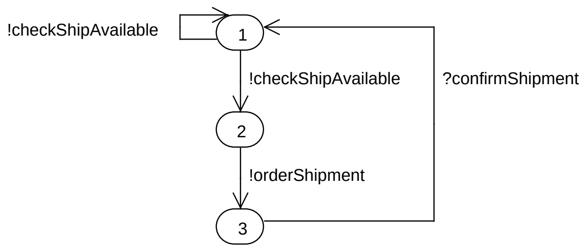

Fig. 6. A basic STS specification for the PSW port

involved. In Fig. 6 the basic STS specification for the `PSW` port is given. This port together with the `PSC` port constitutes the `Supplier` WS. In principle both specifications could serve as the input for two independent test systems, each one testing a single port. But such a setting would ignore the logic combining the two ports. One of the assumptions of MBT is that a monolithic specification of the whole SUT, i.e. the WS protocol and signature, is available. However, when trying to integrate specifications corresponding to ports of a WS we are still leaking information about the dependencies between operations that occur at, for instance, the `PSC` and `PSW` ports. We propose here a research direction of defining service descriptions which allow for an aggregated model of the WS as a whole.

4.3 Specifications Embracing Several Ports

Having access to a service description covering the dependencies between provided and required services we can give a coordination protocol comprising a whole WS. Such an STS needs to make explicit at which port an action occurs. In Figure 7 we do so by preceding each action label by its corresponding port name. The shown STS combines both ports, `PSC` and `PSW`. Such a multi-port STS can then serve as the input for a tester covering all ports of a WS. For instance the upper specification empowers a tester to play both the role of the `Customer` and the role of the `Warehouse` when testing the `Supplier`.

The underlying theory on STSs is not yet ready to deal with such multi-port specifications. Though, there is a version of **ioco** called **mioco** [10] which can deal with several so called channels. One of our intended next steps is to define a similar extension for the STS theory to gain a theoretically sound approach for testing WSs in a coordinated setup.

4.4 Specifying Synchronous and Asynchronous Messages

In Fig. 8 it can be seen why the asynchronous nature of STS is of high value also in a setting of strictly synchronously communicating components. We can explicitly express and test for the order in which operation calls and returns appear. We exemplify the setup again from the view of the `Supplier` WS, receiving a `m1()` call from the `Customer` WS. This call is synchronous, hence

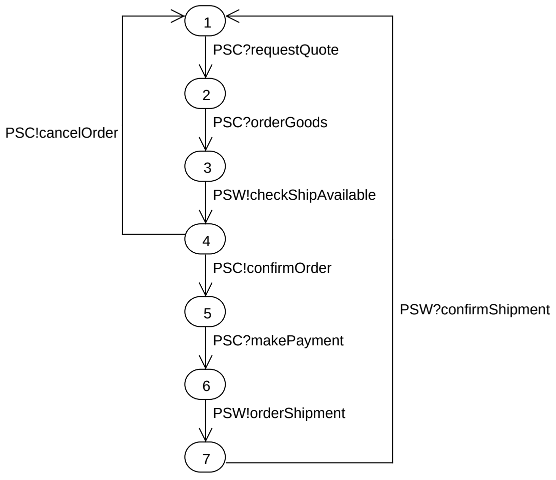

Fig. 7. A basic STS specification for the `Supplier` WS

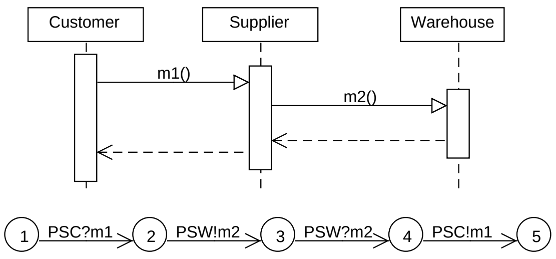

Fig. 8. Modelling synchronous calls with asynchronous actions

during its duration the flow of control of the `Customer` is blocked. This focus of control is visualised in a sequence diagram by a thin rectangle showing the time during which an object is performing an action. During the execution of the `m1()` call the `Supplier` calls the `Warehouse` via the `m2()` operation. Having completed the execution of `m2()` the `Warehouse` sends a proper return message to the `Supplier` which in turn sends a return message to the `Customer`.

The given STS implements this order of operation calls and returns. Due to its asynchronous nature we can disperse input and output events in the shown manner. This is another important motivation to focus on asynchronous models like STSs, especially when dealing with a coordinated setup.

4.5 Input Enabledness

The underlying test theory in our approach (**ioco**) makes some strict assumptions about the implementation under test. One of these assumptions is that the implementation should be input enabled, i.e. under all conditions the system should be able to accept any input. A WS does not satisfy this requirement. Currently, we are revising the **ioco** theory and weaken this requirement. As a consequence we are now able to check whether an input is refused by an implementation. This means that when it is specified that a WS should accept an input message and it does not, we can detect it. Besides from theoretical soundness of applying the theory to WSs, we gain a stricter implementation relation resulting in a more rigorous assessment on the correct implementation of the WS. In other words, we can find more erroneous implementations.

4.6 Testing a Coordination

Finally coming to a coordinated setup comprising several WSs we encounter in principle the same defiances as when generalising from single ports to WSs: integration and more complex PCOs. Again we need a theoretically sound way to integrate service specifications of WSs, and to deal with the demands of a distributed testing scenario.

5 Conclusions and Related Work

When applying well-established theories and tools for MBT of reactive systems in the realm of WSs, one is confronted with the specific originalities of this domain. To analytically approach these new demands we model a setup of communicating WSs as an UML Component Diagram, making the entities explicit which are relevant for establishing communication between WSs. In this setting a set of ports constitutes a WS. Such a port turns out to be the smallest meaningful entity we can identify for serving as a PCO. For specifying such a port the STS formalism is highly suited due to its asynchronous nature. We have also motivated the embedding of synchronous interactions. By so doing we can apply the rich testing theory as described in [7].

A specification of a WS comprising several ports cannot be given as a standard STS anymore since the actions have to be mapped to the ports where they occur. This is more than just a mere technicality since concepts fundamental to the underlying theory like input-enabledness have to be reconsidered. We made these issues explicit to give a clear picture of the problems to be tackled in future research.

Thus, having a complete and formal treatment of testing WSs is still a long way to go. We have proposed a direction which is founded on theories and tools which have proven to be of high value in the domain of reactive systems. As we see it now there is a promising chance that this fundament can be adopted to scale up into the realm of coordinated WSs.

There are other aspects when specifying and testing WSs which might be of high relevance in specific application domains. For instance expressing and testing timing constraints is an obvious candidate here. First promising results in extending the **ioco** theory in this direction have been published [5].

In [3] we have proposed to enrich the interface specification with an UML *Protocol State Machine* (PSM) [12]. We could motivate the transformation of such a PSM into an STS. The whole process was embedded in a framework for WS testing called *Audition* [4]. PSMs basically reflect the need to specify legal orderings of invocations for a specific interface or port. It turned out that mapping a PSM to a strict formal model like an STS is far from being straightforward. It is one branch of our current interests to find an adequate way of using UML models on top of STSs.

In [9] the authors propose to include graph transformation rules that will enable the automatic derivation of meaningful test cases that can be used to assess the behaviour of the WS when running in the "real world". To apply the approach they require that a WS specifically implements interfaces that increase the testability of the WS and that permit to bring the WS in a specific state from which it is possible to apply a specified sequence of tests.

The idea of providing information concerning the right order of the invocations can be found in a different way also in specifications such as BPEL4WS and the Web Service Choreography Interface (WSCI). The use of such information as main input for analysis activities has also been proposed in [8].

References

[1] Alonso, G., F. Casati, H. Kuno and V. Machiraju, "Web Services – Concepts, Architectures and Applications," Springer, 2004.

[2] Belinfante, A., J. Feenstra, R. d. Vries, J. Tretmans, N. Goga, L. Feijs, S. Mauw and L. Heerink, *Formal test automation: A simple experiment*, in: G. Csopaki, S. Dibuz and K. Tarnay, editors, 12^{th} *Int. Workshop on Testing of Communicating Systems* (1999), pp. 179–196.

[3] Bertolino, A., L. Frantzen, A. Polini and J. Tretmans, *Audition of web services for testing conformance to open specified protocols*, in: R. Reussner, J. Stafford and C. Szyperski, editors, *Architecting Systems with Trustworthy Components*, number 3938 in LNCS (2006).

[4] Bertolino, A. and A. Polini, *The audition framework for testing web services interoperability*, in: *Proceedings of the 31st EUROMICRO International Conference on Software Engineering and Advanced Applications*, Porto, Portugal, 2005, pp. 134–142.

[5] Briones, L. B. and E. Brinksma, *Testing multi input-output real-time systems*, in: *ICFEM 2005 Seventh International Conference on Formal Engineering Methods.* (2005).

[6] Christensen, E. et al., *Web Service Definition Language (WSDL) ver. 1.1* (2001).
URL http://www.w3.org/TR/wsdl

[7] Frantzen, L., J. Tretmans and T. Willemse, *Test generation based on symbolic specifications*, in: J. Grabowski and B. Nielsen, editors, *FATES 2004*, number 3395 in LNCS (2005), pp. 1–15.

[8] Fu, X., T. Bultan and J. Su, *Analysis of interacting BPEL web services*, in: *Proc. of WWW2004*, 2004, new York, New York, USA.

[9] Heckel, R. and L. Mariani, *Automatic conformance testing of web services*, in: *Proc. FASE*, Edinburgh, Scotland, 2005.

[10] Heerink, A. W., “Ins and Outs in Refusal Testing,” Ph.D. thesis, University of Twente.

[11] Jard, C. and T. Jéron, *TGV: theory, principles and algorithms*, in: *IDPT ’02* (2002).

[12] Object Management Group, “UML 2.0 Superstructure Specification,” ptc/03-08-02 edition, adopted Specification.

[13] Tretmans, J., *Test generation with inputs, outputs and repetitive quiescence*, Software—Concepts and Tools **17** (1996), pp. 103–120.