Audition of Web Services for Testing Conformance to Open Specified Protocols⋆

Antonia Bertolino[1], Lars Frantzen[2], Andrea Polini[1], and Jan Tretmans[2]

[1] Istituto di Scienza e Tecnologia della Informazione "Alessandro Faedo"
Consiglio Nazionale delle Ricerche
via Moruzzi, 1 – 56124 Pisa, Italy
{antonia.bertolino, andrea.polini}@isti.cnr.it
[2] Institute for Computing and Information Sciences,
Radboud University Nijmegen, The Netherlands
{lf, tretmans}@cs.ru.nl

Abstract. A Web Service (WS) is a type of component specifically conceived for distributed machine-to-machine interaction. Interoperability between WSs involves both data and messages exchanged and protocols of usage, and is pursued via the establishment of standard specifications to which service providers must conform. In previous work we have envisaged a framework for WS testing. Within this framework, this paper focuses on how the intended protocol of access for a standard service could be specified, and especially on how the conformance of a service instance to this specified protocol can then be tested. We propose to augment the WSDL description with a UML2.0 Protocol State Machine (PSM) diagram. The PSM is intended to express how, and under which conditions, the service provided by a component through its ports and interfaces can be accessed by a client. We then propose to translate the PSM to a Symbolic Transition System, to which existing formal testing theory and tools can be readily applied for conformance evaluation. A simple example illustrates the approach and highlights the peculiar challenges raised by WS conformance testing.

1 Introduction

Service Oriented Architecture (SOA) is the emerging paradigm for the realization of heterogeneous, distributed systems, obtained from the dynamic combination of remote applications owned and operated by distinct organizations. Today the Web Service Architecture (WSA) certainly constitutes the most relevant and widely adopted instance of such a paradigm.

A Web Service (WS) is essentially characterized by the capability to "support interoperable machine-to-machine interaction over a network"[5]. This capability is achieved due to the agreement of all major players on the usage of uniform

WS interfaces, coded into the standard machine-processable Web Service Definition Language (WSDL) format [9], and of the Simple Object Access Protocol (SOAP) [18] for WS communication. Moreover, WSA interconnects service providers and service requesters via a standard Service Broker called the UDDI (Universal Description and Discovery Integration)[10]. The information in this catalog follows the yellow, green or white pages paradigms open technology, and defines a common mechanism to publish and retrieve information about available Web Services.

From a methodology viewpoint, WSA builds on the extensive framework of the Component-Based Software Development (CBSD) paradigm, of which it can be considered an attractive successor. Where in fact CBSD pursued the development of a composite system by the assembly of pre-existing (black-box) components, WSA chases the dynamic composition of services at client requests. The two paradigms share the underlying philosophy of developing building blocks (either components or services) of a system for external generalized reuse, whose implementation details are hidden behind a published interface.

By building on the extensive results of CBSD, WSs can today rely on a much more mature culture for compositional development, as testified by the emergence of established standard access and communication protocols. On the other hand, by exacerbating the aspects of loose coupling, distribution and dynamism, WSs have also inherited the most challenging issues of the component-based approach, directly descending here from the need of dynamically composing the interactions between services whose internal behavior is unknown. This fact brings several consequences on the trustability and reliability of WSA; in particular, it calls for new approaches to validate the behavior of black-box components whose services are invoked by heterogeneous clients in a variety of unforeseen contexts.

Although similar problems have been encountered and tackled in the area of software components, testing of WSs is even more difficult since the different machines participating in the interaction could be dispersed among different organizations, so even a simple monitoring strategy or the insertion of probes into the middleware is not generally feasible. Moreover, the notion of the WSA establishes rigid limitations on the kind of documentation that can be provided and used for integrating services. In particular, a service must not include information on how it has been implemented. This obviously is desirable to enable the decoupling between requesters and providers of services, but obviously makes integration testing more difficult.

Speaking in general, it is clear that the capability of testing a software artefact is strongly influenced by the information available [3]. In fact, different kinds of testing techniques can be applied depending on the extent and formalization degree of the information available. The technique to be applied will also be different depending on the quality aspects to be evaluated, e.g. functionality, performance, interoperability, etc.

In CBSD, different proposals have been made to increase the information available with software components [24], following what we generally refer to as the metadata-based testing approach [25]. Fortunately, as already said, today

the area of WS can rely on a more mature attitude towards the need for standardized documentation, with respect to the situation faced by early component developers, and in fact the interaction among WSs is based on a standardized protocol stack and discovery service. Current practice is that the information shared to develop interacting WSs is stored in WSDL files. However, such documents mainly report signatures (or syntax) for the available services, but no information concerning specific constraints about the usage of the described service can be retrieved. Obviously, this way of documenting a service raises problems regarding the capability of correctly integrating different services. In particular, the technology today relies on the restrictive assumption that a client knows in advance the semantics of the operations provided by a service or other properties of it [1].

To facilitate the definition of the collaborations among different services, various approaches are being proposed to enrich the information that should be provided with a WS. Languages such as the Business Process and Execution Language for Web Services (BPEL4WS) and the Web Service - Choreography Description Languages (WS-CDL) are emerging [1], which permit to express how the cooperation among the services should take place. The formalized description of legal interactions among WSs turned out to be instrumental in verifying interoperability through the application of specific conformance evaluation instruments.

We claim that it would be highly useful to attach this description in the form of an XML Metadata Interchange (XMI [29]) file, since in this form it can be easily reused by UML based technologies. XMI is becoming the de facto standard for enabling interaction between UML tools, and it can be automatically generated from widespread UML editors such as IBM Rational Rose XDE or Poseidon.

It is indeed somewhat surprising how two broad standardization efforts, such as the UML and the WSA, are following almost independent paths within distinct communities. Our motivating goal is the investigation of the possibility to find a common ground for both communities. Hence our proposal is that the WS description (including the WSDL file) will report some additional information documented by the WS developer in UML, and in particular, as we explain below, as a Protocol State Machine, that is a UML behavior diagram newly introduced into the latest version of this language [11]. In this way an XMI file representing the associated PSM could be inserted in the UDDI registry along with the other WS documentation. Moreover, as we show in this paper, the PSM provides a formal description of the legal protocol for WS interaction, and following some translation step it can be used as a reference model for test case derivation, applying well established algorithms from formal testing theory.

The framework for automatic testing of WSs presented in this paper has been specifically defined considering technologies related to the WS domain. It will probably be straightforward to apply a similar approach also in a Component Based (CB) setting when the necessary information is provided as data attached to the component. WSs can be considered as being an extreme consequence of

the CB paradigm, in which the developer of a system looses the control, also at run time, of the "assembled" components.

The paper is structured as follows: Section 2 provides an overview of the different flavors of the interoperability notion for WSs, and in particular introduces WS conformance testing; Section 3 presents PSMs and their proposed usage for WS protocol specification; Section 4 synthesizes the general framework we propose for WS testing, and Section 5 outlines related work. In Section 6 a short survey of formal approaches to conformance testing is given, before focusing on the specific formalism which we are going to exploit for WS conformance testing, called Symbolic Transition Systems (STSs). In Section 7 we relate the PSM specification to the presented STS one. Finally, Section 8 summarizes our conclusions and mentions the work remaining to be done.

2 Interoperability of Web Services

Web Services are cooperating pieces of software that are generally developed and distributed among different organizations for machine-to-machine cooperation, and which can act, at different times, as servers, providing a service upon request, or as clients, invoking some others' services. The top most concern in development of WSA is certainly WS interoperability. Actually, WS interoperability is a wide notion, embracing several flavors, all of them important. Without pretending to make a complete classification, for the sake of exposition in our case we distinguish between two main kinds of interoperability issues. A first type of interoperability refers to the format of the information stored in the relevant documents (such as WSDL files, UDDI entry), and to the format of the exchanged SOAP messages. This interoperability flavor is briefly presented below in Section 2.1, in which the approach defined by the WS-I consortium (where the "I" stands for Interoperability) to ensure this kind of interoperability is outlined. A second interoperability issue, discussed in Section 2.2, is instead relative to the correct usage of a WS on the client's side, in terms of the sequencing of invocations of the provided services. Certainly, other kinds of heterogeneity hindering correct interactions of WSs can be identified. For instance, in [16] the authors report about an interesting experience in integrating externally acquired components in a single system. As they highlight, different assumptions made by the different components, such as who has to take the control of the interaction, often prevent real interoperability.

2.1 Data and Messaging Conformance

As said, a first factor influencing the interoperability of WSs is obviously related to the way the information is reported in the different documents (such as SOAP messages, WSDL files, UDDI entries) necessary to enable WS interactions, and to the manner this information is interpreted by cooperating WSs.

This concern is at the heart of the activities carried on by the WS-I consortium, an open industry organization which joins diverse communities of Web

Service leaders interested in promoting interoperability. WS-I provides several resources for helping WS developers to create interoperable Web Services and verify that their results are compliant with the WS-I provided guidelines. In particular, WS-I has recently defined several profiles [12] that define specific relations that must hold among the information contained in different documents. In so doing the interactions among two or more WSs are enabled. Besides, WS-I has defined a set of requirements on how specific information reported in these files must be interpreted.

Just to show the kind of interoperability addressed by WS-I we report below without further explanation a couple of examples from the specification of the Basic Profile. Label R1011 identifies a requirement taken from the messaging part of the profile, which states:

```
R1011 - An ENVELOPE MUST NOT have any element children of soap:
Envelope following the soap:Body element.
```

The second example has been taken from the service description part and describes relations between the WSDL file and the related SOAP action:

```
R2720 - A wsdl:binding in a DESCRIPTION MUST use the part
attribute with a schema type of "NMTOKEN" on all contained
soapbind:header and soapbind:headerfault elements.
```

Alongside the Basic Profile, the WS-I consortium also provides a test suite (that is freely downloadable from the WS-I site) that permits to verify the conformance of a WS implementation with respect to the requirements defined in the profile. In order to be able to verify the conformance of the exchanged messages, part of the test suite acts as a proxy filtering the messages and verifying that the conditions defined in the profile are respected.

The WS-I profile solves many issues related to the representation of data, and to how the same data are represented in different data structures. Another kind of data-related interoperability issue not addressed by the WS-I profile concerns instead the interpretation that different WSs can give for the same data structure. Testing can certainly be the right tool to discover such kind of mismatches. The testing theory presented in Section 6 and its application to the domain of WSs, exerted in Section 7, provides a formal basis for the derivation of test cases that permit to verify that a single implementation of a WS correctly interprets the exchanged data.

2.2   Protocol Conformance

A different interoperability flavor concerns the correct usage of a WS on the client's side, in terms of the sequencing of invocations of the provided services. A correct usage of a WS must generally follow a specified protocol defining the intended sequences of invocations for the provided interface methods, and possibly the pre- and post-conditions that must hold before and after each invocation, respectively.

This is the kind of interoperability we focus on in the remainder of this paper. Generally speaking a protocol describes the rules with which interacting entities must comply in their communication in order to have guarantees on the

actual success of the interaction. Such rules are generally defined by organizations that act as (de facto) standard bodies. The aim of companies joining these organizations is to allow different vendors to produce components that can interact with each other. Often the rules released by such organizations are actually established as standards and adopted by all the stakeholders operating in the interested domain.

It is obvious to everybody though that the definition of a standard per se does not guarantee correct interaction. What is needed in addition is a way to assess the conformance of an implementation to the corresponding specification. Different ways can be explored to verify such conformance, among which testing deserves particular attention. Protocol testing is an example of functional testing in which the objective is to verify that components in the protocol correctly respond to invocations made in correspondence to the protocol specification.

Conformance of an implementation to a specification is one of the most studied subjects from a formal and empirical point of view. Several studies have been carried on for assessing conformance when protocol specifications are considered.

Conformance testing is a kind of functional testing in which an implementation of a protocol entity is solely tested with respect to its specification. It is important to note that only the observable behavior of the implementation is tested. No reference is made to the internal structure of the protocol implementation. In [26] the parties are listed which are involved in the conformance testing process. In a SOA paradigm this can be partially revised in the following way:

1. the implementer or supplier of a service that needs to test the implementation before selling it;
2. the user of a service, claiming to be conform, can be interested in retesting the service to be sure that it can actually cooperate with the other entities already in his/her system;
3. organizations that act as service brokers and that would like to assess the conformance of a service before inserting it in the list of the available services;
4. third parties laboratories that can perform conformance testing for any of the previously mentioned parties.

It is worth noting that the standard may provide different levels of conformance, for instance defining optional features that may or may not be implemented by an organization. This has to be taken in to account when it comes to conformance testing.

3 UML and Web Services

As said in the Introduction, an important topic that is not receiving adequate attention in the research agenda of WS developers and researchers is how the Unified Modeling Language (UML) can be fruitfully used to describe a WS specification and interaction scenarios. The basic idea is to increase the documentation of a WS using UML diagrams. The motivation is to find a trade-off between

a notation which is widespread and industrially usable on one side, but that is also apt to formal treatment and automated analysis on the other. Therefore, a wealth of existing UML editors and analysis tools could be exploited also for WS development. Moreover, from these diagrams a tester should be able to generate useful test suites that, when run on a specific implementation, would provide a meaningful judgment about conformance with the "standard" specification.

The forthcoming UML 2.0 [23] introduces several new concepts and diagrams, in particular supporting the development of Component-Based software. Among these, Protocol State Machine (PSM) diagrams seem particularly promising for our purposes. The idea underneath PSMs is to provide the software designer with a means to express how, and under which conditions, the service provided by a component through its ports and interfaces can be accessed by a client, for instance regarding the ordering between invocations of the methods within the classifier (port or interface). The PSM diagram directly derives from that of the State Machine but introduces some additional constraints and new features. The UML 2.0 Superstructure Specification [23] provides the following definition for this kind of diagram: *A PSM specifies which operations of the classifier can be called in which state and under which condition, thus specifying the allowed call sequences on the classifier's operations. A PSM presents the possible and permitted transitions on the instances of its context classifier, together with the operations which carry the transitions. In this manner, an instance lifecycle can be created for a classifier, by specifying the order in which the operations can be activated and the states through which an instance progresses during its existence.*

Another interesting feature of these diagrams is that they support the definition of *pre-* and *post-conditions* associated with the methods in the interface. This feature provides an improved semantical characterization of the offered services and at the same time increases the verification capability of testers by permitting the application of the well known Design-by-Contract [22] principles. Using first order logic a contract characterizes a service by specifying the conditions that should hold before the invocation and conditions that will be true after the execution of the service. At the same time a contract can specify invariant conditions that remain true during the whole execution of the service. Contracts have proved to be a useful mechanism in CB development, and in particular for the testing of COTS, as for instance developed in [17], and its usage for WS testing is being explored, e.g. [19].

A pre-condition can contain constraints on the parameters passed to the method, or on the values in any way associated to the current status of the environment. If a pre-condition is fulfilled when the invocation is triggered, the implementation must guarantee the respect of the constraints expressed in the post-conditions. In UML this kind of constraints can be naturally expressed using OCL. When a PSM of a WS is defined also using pre- and post-conditions, the assumption is that a component interacting with this WS needs to know these rules, but no more details than these rules, to correctly interact with the system. In fact, a PSM does not define the detailed behavior elaborated inside a component, since this kind of information stays in the scope of traditional State

Machines (or more precisely, Behavioral State Machine - BSM - as defined in UML 2 Superstructure Specification [23]). Instead a state in a PSM represents the point reached in the foreseen interaction between the client and the server. Obviously the definition of a PSM will also influence the definition of the BSM for the object to which the associated port or interface belongs. In order to have a non conflicting specification, related PSMs and BSMs must in some way be compatible.
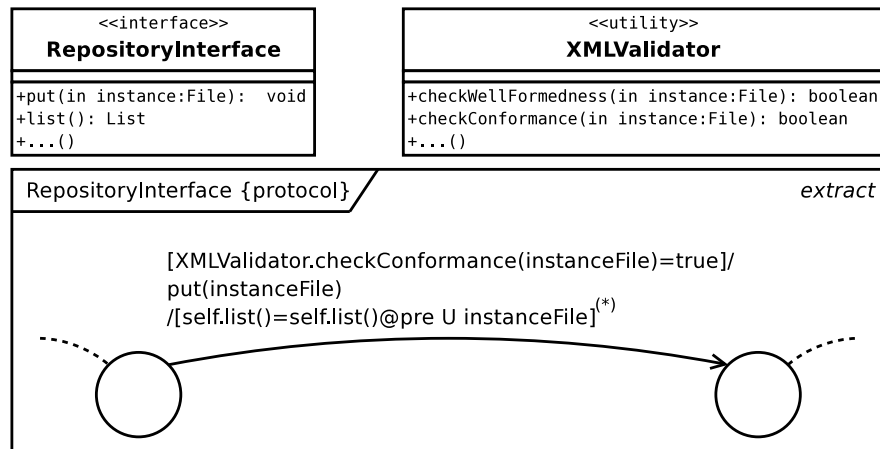
In the specification of a PSM it is important that no assumptions about the real implementation of the classifier are made, for instance it is incorrect to refer, within a pre- or post-condition, to an internal (state-)variable of a possible implementation of the classifier. Nonetheless, when dealing with non-trivial specifications, it is somehow inevitable to (abstractly) model the internal state of the system. This is done via (internal) variables of the specification model. These two views on a system, the implementation and specification view, cause sometimes confusion. For instance, there is no semantical correspondence between the variables of the specification model and the implementation model.

Therefore, one central issue when dealing with PSMs is how to make them completely neutral with respect to internal specification variables. Several ways could be found to deal with this issue. The one we will exemplify in this paper is the augmentation of the classifier with side-effect free "get"-methods to increase its observability. These methods can then be used in the corresponding specification PSM instead of internal variables. On the other hand, such specified "get"-methods must then be implemented. This imposes extra work on the implementer for developing these additional methods, but the advantage of this practice is the possibility of expressing a more precise definition for the implementation, with corresponding benefits regarding conformance evaluation.

It can also be useful to introduce other elements in the model which ease the specification of pre- and post-conditions in the PSM. For instance, having to handle parameters representing XML files it could be useful to introduce a class with methods that can check the well-formedness or the validity of an instance with respect to a corresponding XML Schema. Again this means extra-burden on the side of the developers, because for checking purposes such a `<<utility>>` class needs to be instantiated at run-time. Figure 1, for instance, shows an example of a `<<utility>>` class added in order to facilitate the expression of XML instance conformance within the pre-condition of a generic PSM transaction.

4   A Framework for Web Service Testing

In this section we briefly summarize a framework for testing of WSs, which we have previously introduced in [4]. The framework relies on an increased information model concerning the WS, and is meant for introducing a test phase when WSs ask for being published on a UDDI registry. In this sense we called it the "Audition" framework, as if the WS undergoes a monitored trial before being put "on stage". It is worth noting that from a technical point of view the implementation of the framework does not present major problems, and even from the

<<interface>> **RepositoryInterface**	<<utility>> **XMLValidator**
+put(in instance:File): void +list(): List +...()	+checkWellFormedness(in instance:File): boolean +checkConformance(in instance:File): boolean +...()

RepositoryInterface {protocol}   *extract*

[XMLValidator.checkConformance(instanceFile)=true]/
put(instanceFile)
/[self.list()=self.list()@pre U instanceFile]$^{(*)}$

(*) To simplify we express with the symbol "U" the union of two
"list" sets so avoiding to expand the definition for the type List

Fig. 1. Introduction of a utility class in a model

scientific perspective it does not introduce novel methodologies; on the contrary, one of its targets is to reuse sophisticated software tools (such as test generators) in a new context. The major difficulties we foresee is that a real implementation based on accepted standards requires that slight modifications/extensions are made to such standard specifications as UDDI. This in turn requires wide acceptance from the WS community and the recognition of the importance of conformance testing.

Figure 2 shows the main elements of the framework. The figure provides a logical view, i.e., the arrows do not represent invocations on methods provided by one element, but a logical step in the process; they point to the element that will take the responsibility of carrying on the associated operation.

The process is activated by the request made by the provider of a WS asking for the inclusion of it in the entries of a registry and is structured in eight main steps, which are also annotated in Fig. 2 (numbers in the list below correspond to the numbers in the figure):

1. a Web Service WS1 asks a UDDI registry to be published among the services available to accept invocations;
2. the UDDI service puts WS1 in the associated database, but marking the registration as a pending one, and starts the creation of a specific tester;
3. the WS Testing Client will start to make invocations on WS1, acting as the driver of the test session;
4. during the audition, WS1 will plausibly ask the UDDI service for references to other services;
5. UDDI checks if the service asking for references is in the pending state. If not the reference for the WSDL file and relative binding and access point to the service are provided. In the case that the service is in the pending state
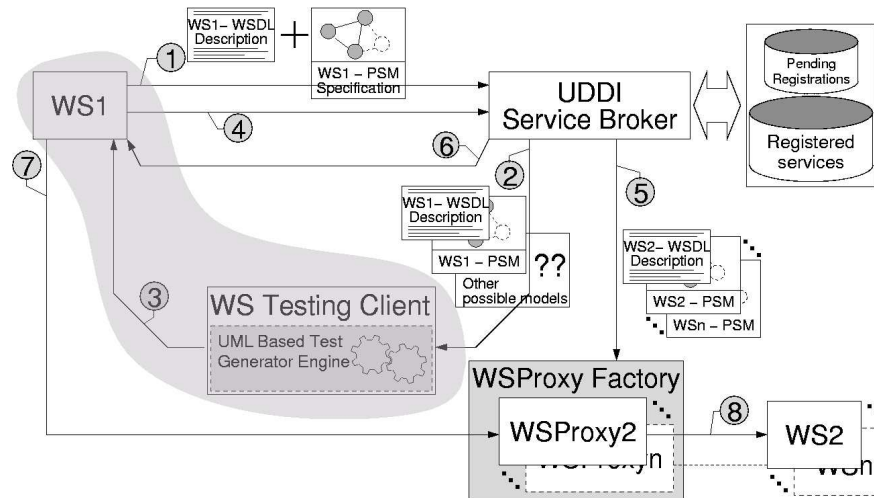
Fig. 2. The Audition Framework

the UDDI will generate, using a WS factory, a WS Proxy for the required services;

6. for each inquiry made by WS1 the UDDI service returns a binding reference to a Proxy version of the requested service;

7. WS1 will start to make invocations on the Proxy versions of the required services. As a consequence the Proxy version can check the content and the order of any invocation made by WS1;

8. if the current invocation is conforming, the Proxy service invokes the real implementation of the service and returns the result obtained to the invoking (WS1) service. Then the process continues driven by the invocations made by the testing client.

In this framework several testing exigencies and approaches can be identified. Specifically, we address the scenario that a standard body has published some WS specification (adopting the PSM notation) and the conformance to this specification of a developed WS instance must be validated. The conformance theory discussed in the following of this paper is applied to the conformance testing of the interface provided by a single Web service. In Fig. 2 the elements involved in this verification have been shaded. As a future step we want deal with extending the existing results from conformance theory to also consider interactions among services. Before presenting our approach we briefly overview recent related work in the field of WS testing.

5 Related Work in Testing of Web Services

WSs testing is an immature discipline in intense need of further research by academy and industry. Indeed, while on the practitioner's side WSs are evi-

dently considered a key technology, research in the area seems not to draw an adequate attention from the testing community, probably due to the contiguity/overlap with other emerging paradigms, especially with Component-Based Software Engineering (CBSE), or perhaps to the quite technical details that this discipline entails. In this section we give a brief overview of those papers that share some similar views with our work.

The possibility of enhancing the functionality of a UDDI service broker with logic that permits to perform a testing step before registering a service has been firstly proposed in [28] and [27], and subsequently in [20]. This idea is also the basis for the framework introduced in this paper. However, the information we use and the tests we derive are very different from those proposed in the cited papers. In particular while in the cited works testing is used as a means to evaluate the input/output behavior of the WS that is under registration, in the Audition framework we mainly monitor the interactions between the WS under registration with providers of services already registered. In this sense, we are not interested in assessing if a WS provided is bug-free in its logic, but we focus on verifying that a WS can correctly cooperate with other services, by checking that a correct sequence of invocations to the service leads in turn to a correct interaction of the latter with other services providers (and that vice versa an incorrect invocation sequence receives an adequate treatment).

With reference to the information that must be provided with the WS description, the authors of [28] foresee that the WS developer provides precise test suites that can be run by the enhanced UDDI. In [20] instead 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 behavior 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 [15]. However, the objective of the authors in this case is to formally verify that some undesired situations are not allowable given the collaboration rules. To do this the authors, after having translated the BPEL specifications into Promela (a language that can be accepted by the SPIN model checker), apply model checking techniques to verify if specific properties are satisfied. Another approach to model-based analysis of WS composition is proposed in [13]. From the integration and cooperation of WSs the authors synthesize Finite State Machines and then they compare if the obtained result and allowable traces in the model are compatible with that defined by BPEL4WS-based choreography specification.

6 Model-Based Conformance Testing

As said, conformance verification involves both static conformance to an established WSDL interface, and dynamic conformance of exposed behaviors to an established interaction protocol. Clearly the second aspect is the challenging one. In the following we first introduce the basic notion of formal conformance testing and then develop a possible strategy for formal conformance testing of Web Services based on the existing **ioco** implementation relation (see below), and related tools.

We want to test conformance with respect to a protocol specification, given as a PSM. One can see such a PSM as a high-level variant of a simple state machine, such as a Finite State Machine (FSM) or a Labeled Transition System (LTS). Hence we can use classical testing techniques based on these simple models to test for conformance. In this paper we focus on LTS-based testing techniques.

6.1 LTS-based Testing

Labeled Transition Systems serve as a semantic model for several formal languages and verification- and testing theories, e.g. process algebras and statecharts. They are formally defined as follows:

Definition 1. *A* Labeled Transition System *is a tuple* $\mathcal{L} = \langle S, s_0, \Sigma, \rightarrow \rangle$, *where*

- S *is a (possibly infinite) set of* states.
- $s_0 \in S$ *is the* initial state.
- Σ *is a (possibly infinite) set of* action labels. *The special action label* $\tau \notin \Sigma$ *denotes an* unobservable *action. In contrast, all other actions are* observable. Σ_τ *denotes the set* $\Sigma \cup \{\tau\}$.
- $\rightarrow \subseteq S \times \Sigma_\tau \times S$ *is the* transition relation.

In formal testing the goal is to compare a specification of a system with its implementation by means of testing. The specification is given as a formal model in the formalism at hand, so in our case as an LTS or as an expression in a language with underlying LTS semantics. This formal specification is the basis to test the implementation (System Under Test – SUT). This implementation, however, is not given as a formal model but as a real system about which no internal details are known (hidden in a "black box"), and on which only experiments, or tests, can be performed. This implies that we cannot directly define a formal implementation relation between formal specifications and physical implementations. To define such an implementation relation, expressing which implementations are conforming, and which are not, we need an additional assumption, referred to as the *test hypothesis*. The test hypothesis says that the SUT exhibits a behavior which *could* be expressed in some chosen formalism. Now conformance can be formally defined by an implementation relation between formal specification models and assumed implementation models.

Typically, in LTS testing the formalism assumed for implementations is the LTS formalism itself. So, an implementation is assumed to behave as if it were
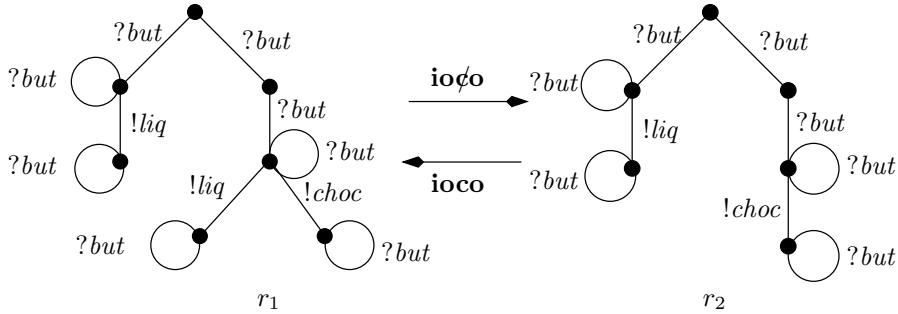
Fig. 3. The **ioco** relation

an LTS. Based on this assumption many different implementation relations on LTSs have been defined in the literature, most of them being preorders or equivalences on LTSs. These relations differ, for example, in the way they treat nondeterminism, the extent to which they allow partial specifications, whether they allow "look-ahead" in behaviors, etc. For several of these implementation relations also testing scenarios and test generation algorithms have been published. These testing algorithms are usually proved to be complete in the limit, i.e., the execution of all (usually infinitely many) test cases constitutes a decision procedure for the relation. For an annotated bibliography for testing based on LTSs see [6], for a survey on existing test theory in general see [8].

The ioco Implementation Relation — In this paper we use the **ioco** implementation relation for testing, see [26]. **ioco** is not a preorder relation, but it turns out to be highly suited for testing. Several state-of-the-art testing tools nowadays implement it, e.g. TorX [2] and the TGV-based tools [21].

In the **ioco** setting the specifications are LTSs where the set of action labels is partitioned into input- and output action labels. The test hypothesis is that implementations can be modeled as *input-output transition systems* (IOTS). IOTSs are a subclass of LTSs for which it is assumed that all input actions are enabled in all states (input enabledness). A *trace* is a sequence of observable actions, starting from the initial state. As an example take the LTSs from Fig. 3. We have one input action *?but*, standing for pushing a button on a machine supplying chocolate and liquorice. These are the output actions *!choc* and *!liq*. Both r_1 and r_2 are input enabled, at all states it is possible to push the button, hence both LTSs are also IOTSs. Some traces of r_1 are *?but·!liq*, *?but·?but·!choc*, and so on.

A special observation embedded in the **ioco** theory is the observation of *quiescence*, meaning the absence of possible output actions. The machine can not produce output, it remains silent, and only input actions are possible. For instance both r_1 and r_2 are quiescent in the initial state (the upmost state), waiting for the button to be pressed. After applying *?but* to the systems, both may be quiescent due to nondeterminism (following the right branch). They

may also nondeterministically chose the left branch and produce liquorice via the output action !*liq*. Hence, when it comes to test generation, and the tester observes quiescence after pushing the button, it knows that the systems chose the right branch, waiting for the button to be pushed again, and can forget about the left branch of the specification. This waiting for output before generating the next input is the principle of *on-the-fly* testing, which helps in avoiding a state space explosion when computing test cases out of a given specification. The test tool *TorX* implements a test generation algorithm which tests for **ioco**-correctness via such an *on-the-fly* approach, see [2]. The observation of quiescence is embedded in the notion of traces, leading to so called *suspension traces*.

We will not give a formal definition of the **ioco** relation here to keep an introductory flavor. We refer to [26] for a detailed description and give instead an informal intuition of it. Let i be an implementation IOTS and s be an LTS specification of it. Then we have:

   i **ioco**-conforms to s $\Leftrightarrow$
   - if i produces output x after some suspension trace σ, then s can produce x after σ
   - if i cannot produce any output after suspension trace σ, then s can reach a state after σ where it cannot produce any output (quiescence)

The addition of quiescence increases the discriminating power of **ioco**, as illustrated by Fig. 3. Taking r_2 as the specification for r_1, we have that r_1 can produce !*liq* and !*choc* after the suspension trace ?*but* · *quiescence* · ?*but*. The specification though can only produce !*choc* after pressing the button, observing quiescence, and pressing the button again. Hence the **ioco** condition "if i produces output x after suspension trace σ, then s can produce x after σ" is not fulfilled, r_1 is not **ioco**-conform to r_2. On the other hand, r_2 is **ioco**-conform to r_1.

Extensions of ioco — The model of LTSs has the advantage of being simple and highly suited for describing the functional behavior of communicating, reactive systems. But in practice one does not want to specify a system in such a low-level formalism. Instead high-level description languages like statecharts, PSMs, or process algebras with data extensions are the preferred choice. Furthermore, non-functional properties like embedding timing constraints into the model are mandatory means in certain application areas. Recent research activities have produced first promising results in dealing with these extensions, see e.g. [7].

In our setting we are interested in testing based on automata allowing for a symbolic treatment of data, meaning that one can specify using data of different types, guarded transitions, and so on. This makes the specification much more compact and readable. At first sight using such formalisms for **ioco** testing is straightforward, because usually these models have an underlying LTS semantics, meaning that one just has to convert the high-level model into its underlying LTS, and then test as usual based on that. The difficulty here is,

that even a small, finite symbolic model has commonly an infinite underlying LTS semantics. This problem is commonly known as *state-space explosion*. To address this problem recent research has focused on testing based directly on a symbolic specification, without mapping it at all to its underlying LTS. The **ioco** relation has been recently lifted to such a symbolic setting based on so called *Symbolic Transition Systems* (STSs), see [14]. Such an STS has many similarities with formalisms like PSMs, and hence serves as a promising choice for testing WSs specified with PSMs.

Symbolic Transition Systems — STSs extend LTSs by 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 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 LTS-semantics of STSs due to its extent, and give instead a simple example in Fig. 4 showing all necessary ingredients. For a formal definition we refer to [14].

The shown STS specifies a counter consisting of three so called locations (the black dots). The uppermost one is the initial location, distinguished by the sourceless arrow. Being in this initial location the system is quiescent, it awaits an input called `?start<>`, which has no parameters. For instance one can think of it as a button. After this button is pressed, the system nondeterministically chooses the left or right branch (called "switches"), and sets the internal variable `x` to zero. Both switches are always enabled because both are unconstrained, the guards say `true`. Each switch 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. As commonly done we precede the names of input actions with a question mark, and the names of output actions with an exclamation mark.

If the system chooses the left switch it first performs an output via the action `!display<n>`. This action has one parameter called `n`, which is constrained in the guard `[n = x]`, saying that the display has to show the value of the internal variable `x` (which is zero at first). Next it increments `x` by one, denoted `x++`, and loops. The right branch does the same, but decrements `x` in every loop. Hence think of `x` and `n` as being declared as integer in the underlying first order structure. Altogether we have a system which, after a button has been pressed, either displays `0,1,2,...`, or `0,-1,-2,...` on a display. Another feature not shown is the use of internal so called τ-switches, which become for instance important when composing systems.

In the next section we give a more complex STS, on which we will also exemplify the **ioco** test generation algorithm.
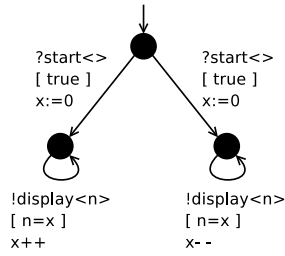
Fig. 4. An STS counter

7 Testing Based on PSMs

In this section we want to provide an idea of how WS conformance testing can be based on PSM specifications using the STS testing approach. PSMs serve as formal specifications of WS behavior. These PSMs are transformed into STSs, which, in turn, have an LTS semantics. This allows us to formally root our work in the **ioco**-testing theory as applicable to STSs [14]. The choice of STSs as the formal notation to be used for the derivation of test cases has been mainly influenced by the expressive power of such a formalism which in principle allows to easily embed in an STS diagram all parts of the information that can be found in a PSM diagram.

A practical example can ease the explanation of the approach that we intend to pursue. We first introduce a PSM example in subsection 7.1. Then we present, informally, the transformation of the PSM into an STS in subsection 7.2. Finally, we exemplify the test case generation in subsection 7.3.

7.1 An Example PSM

Fig. 5 shows a PSM for a Web Service dispensing coffee and tea, for simplicity without giving back the possible change. Obviously this is just a toy-example to illustrate our ideas. However, the coffee machine example already exemplifies most of the features of a Web Service specification for which the protocol can be dependent on the data provided by the client. In fact the drinks can be provided only after the specified amount of money has been inserted.

Each invocation of a method will return an object of type `Status` representing all possible output actions which can be observed by the test system. In particular there is a variable `CoffeeButtonLight` expressing the status of the coffee button, a variable `TeaButtonLight` expressing the status of the tea button, a variable `Display` expressing the value reported by the display, and finally a variable `Drink` expressing the possible drink to be dispensed.

Each transition in the PSM is a tuple of the form (actual-state, pre-condition, method invocation, post-condition, final state). The pre-condition expresses under which constraint a transition is admissible from the given source state.
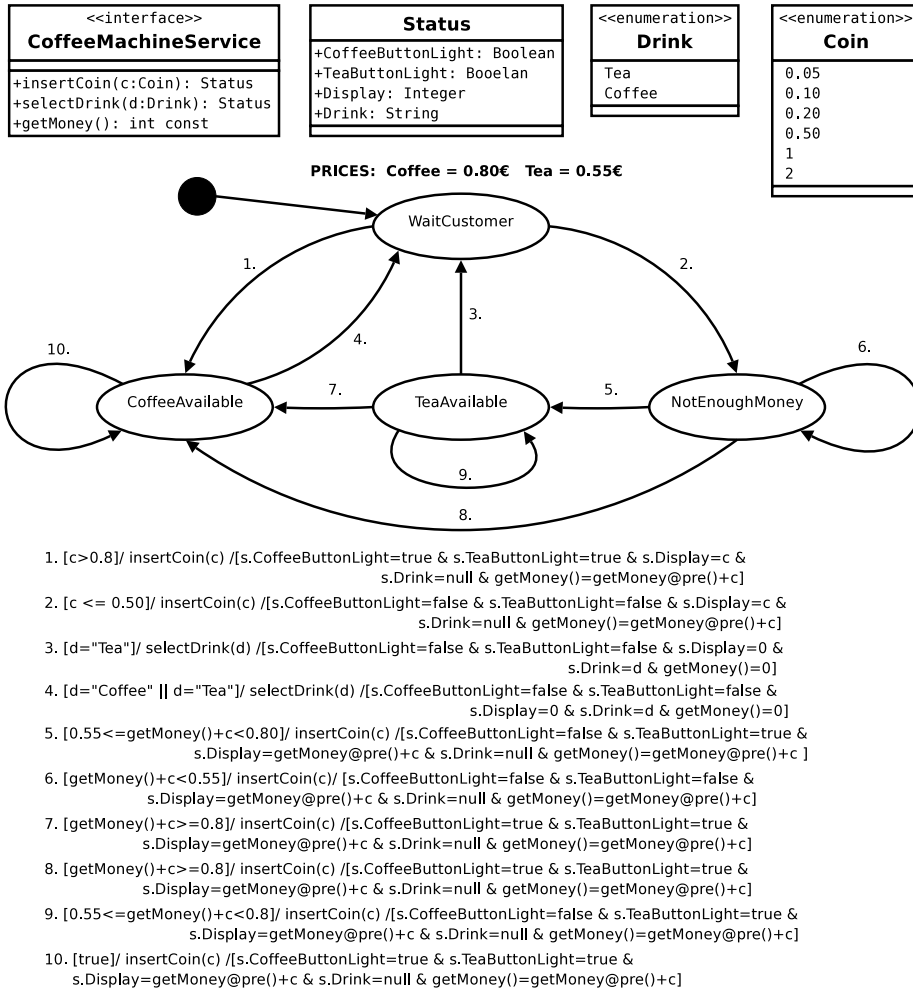
<<interface>> **CoffeeMachineService**	**Status**	<<enumeration>> **Drink**	<<enumeration>> **Coin**
+insertCoin(c:Coin): Status +selectDrink(d:Drink): Status +getMoney(): int const	+CoffeeButtonLight: Boolean +TeaButtonLight: Booelan +Display: Integer +Drink: String	Tea Coffee	0.05 0.10 0.20 0.50 1 2

PRICES: Coffee = 0.80€   Tea = 0.55€

1. [c>0.8]/ insertCoin(c) /[s.CoffeeButtonLight=true & s.TeaButtonLight=true & s.Display=c & s.Drink=null & getMoney()=getMoney@pre()+c]

2. [c <= 0.50]/ insertCoin(c) /[s.CoffeeButtonLight=false & s.TeaButtonLight=false & s.Display=c & s.Drink=null & getMoney()=getMoney@pre()+c]

3. [d="Tea"]/ selectDrink(d) /[s.CoffeeButtonLight=false & s.TeaButtonLight=false & s.Display=0 & s.Drink=d & getMoney()=0]

4. [d="Coffee" || d="Tea"]/ selectDrink(d) /[s.CoffeeButtonLight=false & s.TeaButtonLight=false & s.Display=0 & s.Drink=d & getMoney()=0]

5. [0.55<=getMoney()+c<0.80]/ insertCoin(c) /[s.CoffeeButtonLight=false & s.TeaButtonLight=true & s.Display=getMoney@pre()+c & s.Drink=null & getMoney()=getMoney@pre()+c]

6. [getMoney()+c<0.55]/ insertCoin(c)/ [s.CoffeeButtonLight=false & s.TeaButtonLight=false & s.Display=getMoney@pre()+c & s.Drink=null & getMoney()=getMoney@pre()+c]

7. [getMoney()+c>=0.8]/ insertCoin(c) /[s.CoffeeButtonLight=true & s.TeaButtonLight=true & s.Display=getMoney@pre()+c & s.Drink=null & getMoney()=getMoney@pre()+c]

8. [getMoney()+c>=0.8]/ insertCoin(c) /[s.CoffeeButtonLight=true & s.TeaButtonLight=true & s.Display=getMoney@pre()+c & s.Drink=null & getMoney()=getMoney@pre()+c]

9. [0.55<=getMoney()+c<0.8]/ insertCoin(c) /[s.CoffeeButtonLight=false & s.TeaButtonLight=true & s.Display=getMoney@pre()+c & s.Drink=null & getMoney()=getMoney@pre()+c]

10. [true]/ insertCoin(c) /[s.CoffeeButtonLight=true & s.TeaButtonLight=true & s.Display=getMoney@pre()+c & s.Drink=null & getMoney()=getMoney@pre()+c]

Fig. 5. PSM model for the coffee machine example

For our intended mapping to STSs it is crucial that all method-calls in a pre-condition are query-methods, i.e., they don't affect the internal state of the system (no side effects). The post-condition expresses the expected results of the corresponding method invocation, leading to the given target state. For instance transition 1 declares that entering a coin c greater then 0.80, i.e., 1 or 2 Euro, when being in state WaitCustomer, the result should be the highlighting of the coffee and tea buttons, the visualization of amount "c" in the display, no dispense of any drink, and finally an update of the internal state via the invocation of the getMoney() method. The arriving state for the transition will be CoffeeAvailable. A state in a PSM usually corresponds to a quiescent state of the system, i.e., the system waits for an input from its environment.

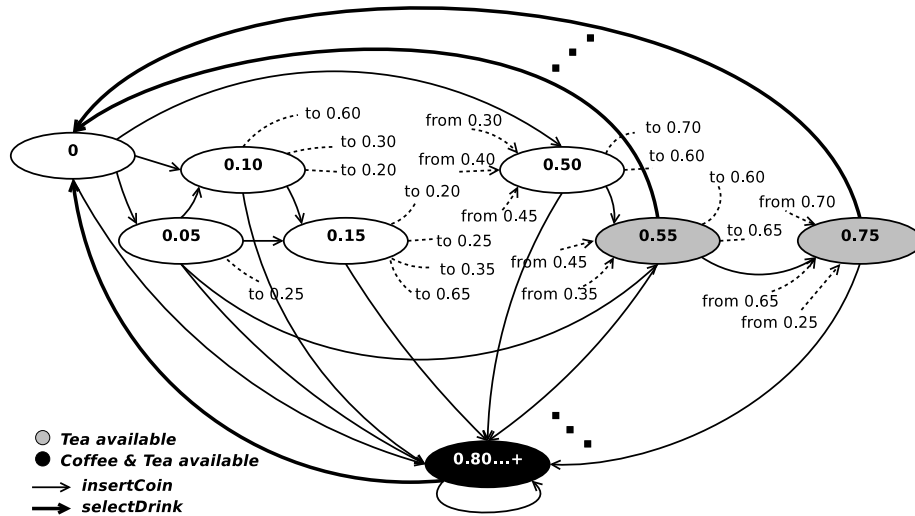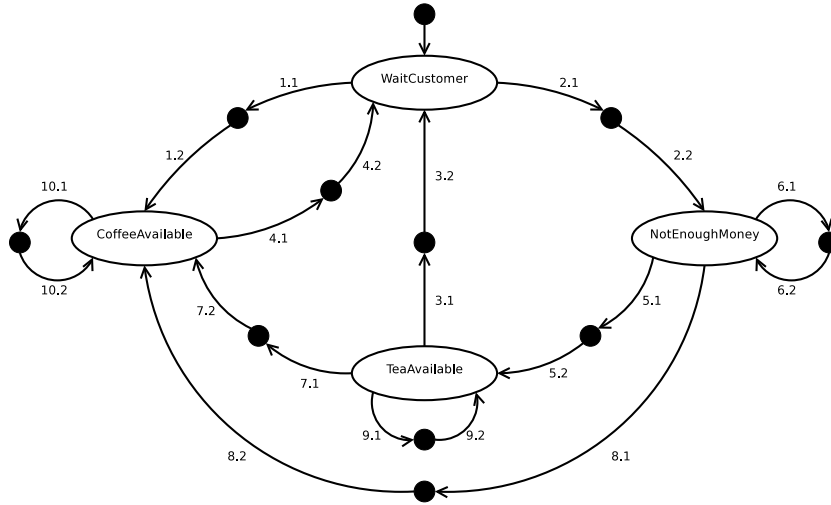**Fig. 6.** PSM and the state space explosion problem

It is worth noting that the model only specifies the correct sequences of method-invocations. Nevertheless, since a client of a WS can invoke the methods in the interface in any order, it becomes important to specify the behavior also when an incorrect sequence is triggered. This corresponds to the input-enabledness as introduced in section 6, and would require the introduction of exceptions to the specification for each method in the interface. In particular for the coffee machine example the invocation of a method in an incorrect order, such as the pushing of the dispense button before it becomes highlighted, should leave the protocol in the same status, notifying the client with an exception. Such a behavior can be considered similar to that of a real coffee machine that triggers a beep when a button is incorrectly pushed. However, since the introduction of exceptions would result in an unwieldy increase in the complexity of the diagram we decided to elide them in the model.

Our simple example shows the influence of the special "get"-method in the interface, related to state data that influences protocol transitions, in this case the `getMoney()` method. Fig. 6 shows an extract of the resulting PSM when no such method is provided (omitting return values). In particular the whole machine contains 17 states and 70 state transitions, a clear complexity increase with respect to the machine in Fig. 5. Having the possibility of logically expressing values through the usage of the `getMoney()` method permits to produce manageable models.

7.2 Mapping from PSM to STS

In Fig. 7 the STS variant of the coffee machine example of Fig. 5 is shown. It consists of 14 locations and 20 switches. The (linear) increase in the number of loca-

WaitCustomer

1.1   2.1

1.2   4.2   3.2   2.2

10.1   CoffeeAvailable   4.1   NotEnoughMoney   6.1

10.2   7.2   3.1   5.1   6.2

7.1   TeaAvailable   5.2

8.2   9.1   9.2   8.1

```
1.1:  ?insertCoin<c> [c>0.8]  c_var:=c
1.2:  !insertCoin<s> [s.CoffeeButtonLight=true & s.TeaButtonLight=true &
                      s.Display=c_var & s.Drink=null]  money+=c_var
2.1:  ?insertCoin<c> [c<=0.5]  c_var:=c
2.2:  !insertCoin<s> [s.CoffeeButtonLight=false & s.TeaButtonLight=false &
                      s.Display=c_var & s.Drink=null]  money+=c_var
3.1:  ?selectDrink<d> [d="Tea"]  d_var:=d
3.2:  !selectDrink<s> [s.CoffeeButtonLight=false & s.TeaButtonLight=false &
                      s.Display=0 & s.Drink=d_var]  money:=0
4.1:  ?selectDrink<d> [d="Coffee"]  d_var:=d
4.2:  !selectDrink<s> [s.CoffeeButtonLight=false & s.TeaButtonLight=false &
                      s.Display=0 & s.Drink=d_var]  money:=0
5.1:  ?insertCoin<c> [0.55<=money+c<0.8]  c_var:=c
5.2:  !insertCoin<s> [s.CoffeeButtonLight=false & s.TeaButtonLight=true &
                      s.Display=money+c_var & s.Drink=null]  money+=c_var
6.1:  ?insertCoin<c> [money+c<0.55]  c_var:=c
6.2:  !insertCoin<s> [s.CoffeeButtonLight=false & s.TeaButtonLight=false &
                      s.Display=money+c_var & s.Drink=null]  money+=c_var
7.1:  ?insertCoin<c> [money+c>=0.8]  c_var:=c
7.2:  !insertCoin<s> [s.CoffeeButtonLight=true & s.TeaButtonLight=true &
                      s.Display=money+c_var & s.Drink=null]  money+=c_var
8.1:  ?insertCoin<c> [money+c>=0.8]  c_var:=c
8.2:  !insertCoin<s> [s.CoffeeButtonLight=true & s.TeaButtonLight=true &
                      s.Display=money+c_var & s.Drink=null]  money+=c_var
9.1:  ?insertCoin<c> [0.55<=money+c<0.8]  c_var:=c
9.2:  !insertCoin<s> [s.CoffeeButtonLight=false & s.TeaButtonLight=true &
                      s.Display=money+c_var & s.Drink=null]  money+=c_var
10.1: ?insertCoin<c> [true]  c_var:=c
10.2: !insertCoin<s> [s.CoffeeButtonLight=true & s.TeaButtonLight=true &
                      s.Display=money+c_var & s.Drink=null]  money+=c_var
```

Fig. 7. STS model for the coffee machine example

tions and switches is due to the fact that STSs are more fine-grained than PSMs, which subsume a guarded procedure call together with its post-conditions on one transition. In STSs a transition corresponds to either an input action (i.e. a procedure call), or an output action (i.e. the returned value of the procedure). Hence every interface method invocation corresponds to an input action and an output

action. For instance, the `insertCoin(c: Coin): Status` method is mapped to an input action `?insertCoin<c: Coin>`, and an output action `!insertCoin<s: Status>`. This allows to specify an asynchronous message interchange, and has a number of consequences, for instance we have to remember the values given to procedures. To do so we store them in variables, an inherent concept of STSs. For instance take transition `1` of the PSM from Fig. 5. This transition is mapped to the transitions `1.1` and `1.2` in Fig. 7. Here the method invocation (i.e. input action) `insertCoin(c)` is executed with the restriction that `c>0.8`. We do so via the guarded transition `1.1`, and store the parameter value `c` in an internal variable `c_var` (called location variable in STSs) to remember it. Next the SUT performs an output action, it returns a `Status` with certain settings. This is done via transition `1.2`. Here we make use of the remembered parameter value of the preceding procedure call by referring to the internal variable via `s.Display=c_var`.

The challenging issue is the mapping of the special "get"-methods in the interface to STSs. The use of these methods corresponds to location variables in STSs. They are used to model state-dependent behavior, and they can be utilized in guards. In our example this concerns the `getMoney()` method, which is mapped onto a location variable `money`. After having inserted a coin, like in transition `1.1`, we have to update the internal state. In the PSM this is expressed via `getMoney()=getMoney@pre()+c`. In the STS we can express this as `money := money + c_var`, shortly written as `money += c_var` in transition `1.2`.

In so doing we can map the PSM into the given STS, which in turn allows us to use the existing **ioco**-based test theory and algorithm, which was adapted for STSs in [14].

Note that this transformation might not always be as easy as in the given example. One problem is that it is possible to partially specify methods like the `getMoney()` in a PSM. For instance we could have simply left out the update `getMoney()=getMoney@pre()+c` when applying the `insertCoin(c)` method in the PSM. Doing so we could not have developed the given STS without applying knowledge about this method which is not in the PSM. Hence in future research we will try to give exact restrictions to PSMs allowing for a guaranteed and automated translation process.

7.3 Automated Testing Based on STSs

In the remainder of this section we give a simple example run of the test generation algorithm as given in [14], which tests for **ioco**-conformance. It generates and executes test cases on-the-fly. That means that instead of firstly computing a set of test cases from the STS, and then applying them to the SUT, it generates a single input, applies it to the SUT, and continues w.r.t. the observed response of the system. As a consequence we avoid the state space explosion when generating test cases, see also [2].

First of all the system has to be initialized by giving initial values to the internal variables, in this case `money` and `c_var`. We assume both to be zero, i.e. no coin has been entered in the coffee machine, yet. At first the initial semantical

state is computed. Such a state consists of a set of locations in which the SUT could currently be, and a valuation of the internal variables. In our case the STS is fully deterministic, therefore the set of locations will always be a singleton. The initial state here is (`WaitCustomer`, (`money=0`, `c_var=0`)).

The basic principle of the algorithm is to continously choose nondeterministically one out of these three options:

- Stop testing and give the verdict `Pass`
- Give an input to the SUT
- Observe output (including quiescence) of the SUT (which may result in `Fail`)

Let's assume that first an input is given. The only specified input action in the initial location is `?insertCoin<c>`. The input constraint is computed for this action, which is a first order formula telling the condition for the parameters under which the action can be applied. To do so all outgoing switches with this specific action have to be taken into account. We get here (`c>0.8`)∨(`c≤0.5`). If a solution exists, one is chosen and applied to the system, e.g. `?insertCoin(0.5)`. Now the set of next possible locations is computed, which is only the one location where transition `2.1` leads. The new values of the variables are (`money=0`, `c_var=0.5`). Now the tester observes output, it receives the returned `Status` object saying that no drink is available and that the display shows `0.5`. This is conformant with the specification, `money` is updated to `0.5` and we proceed to the semantical state (`NotEnoughMoney`, (`money=0.5`, `c_var=0.5`)). If we would have observed a different result, for instance a different display value, the test would have stopped with verdict `Fail`. Choosing next for another `?insertCoin<c>` action we get the input constraint (`0.55≤0.5+c<8`)∨(`0.5+c<0.55`)∨(`0.5+c≥0.8`), assembled from switches `5.1`, `6.1` and `8.1`. Again one solution is chosen for `c`, e.g. `1`. We apply `insertCoin(1)` and observe in the returned status that coffee and tea are available, we end up in state (`CoffeeAvailable`, (`money=1.5`, `c_var=1`)). Now the algorithm may choose to stop the testing and give the verdict `Pass`. 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.

For the audition of WSs it remains to be evaluated how such a 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 (locations, switches, evaluation criteria for the guards, etc.). Also more specific testing scenarios (called test purposes) might be of high value.

In Fig. 8 you find the test case corresponding to the exerted test run. We have abbreviated the returned `Status` object and we give it as a tuple representing the values of `CoffeeButtonLight`, `TeaButtonLight`, `Display`, and `Drink`, respectively. The abbreviation "`qui`" stands for observed "quiescence". As seen in the figure a test case formally corresponds to a tree-like LTS with distinguished terminal states `Pass` and `Fail`. When observing outputs such a test case must tell the tester how to proceed w.r.t. all possible outputs. For instance in our
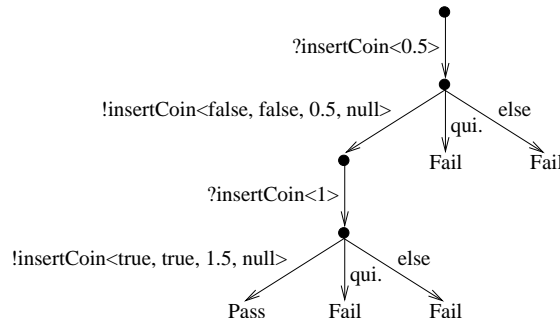
Fig. 8. An example test case

case the test case must specify this for all possible resulting `Status` objects. We have abbreviated this by the usage of an `else` transition. Given a system with a huge (let alone infinite) set of possible output actions, such a test case generation leads to an explosion of the state space. Due to the on-the-fly testing it is not necessary to generate such a complete test-case tree out of the specification, the tester just checks the single observed output for conformance, and continues accordingly.

This simple example does not reveal the hidden complexity within this process, like nondeterminism, or checking for quiescence. It is just presented to exemplify the basic principle. The detailed algorithmical tasks can be found in [14].

8 Conclusions and future work

Our research addresses the problem of testing a WS instance for conformance to a published specification, which could, for instance, be included in the UDDI registry. We have in fact conceived a possible framework for trustable WSs [4], in which a WS before UDDI registration undergoes sort of an "audition" to ascertain both whether it behaves conforming to the specifications when invoked, and also whether it in turn correctly invokes other published services.

The idea is that for widespread services within an application domain, the community agrees on some standard features and functions to be provided from any service provider who wishes to claim conformance to that standard. In this way interoperability between services provided by different companies can be achieved, and this is somehow what is being done by the WS-I initiative relatively to data and messaging conformance. Another important aspect of WS interoperability concerns how the service is used, i.e., the correct sequencing of invocations of the provided WS interface methods, and possibly the pre- and post-conditions to be guaranteed before and after each invocation. Hence, a key open issue in WSA research is currently how to augment the WSDL definition, so to provide a description of the intended usage for a "standard" service.

To obtain WS interoperability establishing a standardized protocol of usage *per se* is not enough: we also need sound approaches to assess that a WS implementation actually conforms to the corresponding standard specification. This is precisely the objective pursued here: we proposed in particular to exploit the extensive background in formal conformance testing of reactive systems, by adapting it to the peculiar features of WSs. On the other hand, to foster industrial adoption, we intend to start from a protocol specification written in the widespread UML notation. In particular, we have identified the PSM diagram of the UML2.0 as a suitable formalism for expressing how a WS has to be accessed. Then, to be able to readily apply the existing algorithms and tools for conformance testing, we envisaged to convert the PSM specification to a Symbolic Transition System model, which in principle possesses an adequate expressive power. Once the STS is derived, we intend to directly apply the test generation algorithm given in [14] which tests for **ioco**-conformance. As discussed, the advantage of this algorithm is that it generates and executes test cases on-the-fly, thus preventing state space explosion.

In this paper we have provided a preliminary overview of the approach and illustrated it through a simple example of a hypothetical coffee dispenser machine (admittedly coffee remains something quite difficult to produce via Internet, but the example is just to be seen as an intuitive illustration of client-server interaction). The latter was already sufficient to highlight the crucial point in the approach we propose: how to specify protocols of interaction between services without making any assumption on the internal implementation of the specific service instances.

We will continue investigating the specific issues raised by WS conformance in general, and the application of the **ioco**-test theory to it in particular. There are several issues which require further investigation. First, the use of the special "get"-methods to model internal state variables extend the visible interface, and they moreover put a requirement on the implementers to implement them correctly, and on the testers to test them. A question is whether there are alternatives to specify this in PSMs, e.g., using something analogous to location variables in STS. Second, a restriction of the formal testing approach currently is that only the providing interface of a WS is tested, and not the invocations to other WSs. Using the PSMs of the invoked services it seems possible to also consider the conformance of these invocations, both in isolation, or in combination with its own PSM. Third, a theoretical question is to what extent the test hypothesis that SUTs behave as input-output transition systems, really hold: can all methods always be invoked at any time? Finally, the translation from PSM to STS should, of course, be generalized, and automated in a tool.

Trustable services are the ultimate goal of our research: we wish to increase the interoperability and testability of WSA by fostering the application of rigorous model based testing methodologies. At present, a huge effort is taken by the various communities towards identifying common standard models for WSs,

allowing for a smooth combination and inter-operation of WSs[3]. It is important to raise awareness within the same communities that also common standard methods for rigorous verification and validation of functional and non-functional properties of WS must be sought. In this sense, we hope that the approach proposed in this paper, although preliminary, provides first convincing arguments and interesting directions for further investigations.

References

1. G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services – Concepts, Architectures and Applications*. Springer Verlag, 2004.
2. A. Belinfante, J. Feenstra, R.G. de 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, 12th *Int. Workshop on Testing of Communicating Systems*, pages 179–196. Kluwer Academic Publishers, 1999.
3. A. Bertolino. Knowledge area description of software testing. In *Guide to the Software Engineering Body of Knowledge SWEBOK*. IEEE Computer Society, 2000.
4. A. Bertolino 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*, pages 134–142, Porto, Portugal, August 30th - September 3rd 2005.
5. D. Booth et al. Web Services Architecture. http://www.w3.org/TR/ws-arch/, February 2004.
6. E. Brinksma and J. Tretmans. Testing Transition Systems: An Annotated Bibliography. In F. Cassez, C. Jard, B. Rozoy, and M. Dermot, editors, *Proceedings of the 4th Summer School on Modeling and Verification of Parallel Processes (MOVEP 2000)*, volume 2067 of *LNCS*, pages 187–195. SV, 2001.
7. L. Brandán Briones and E. Brinksma. A test generation framework for quiescent real-time systems. In J. Grabowski and B. Nielsen, editors, *Formal Approaches to Software Testing, FATES*, Linz, Austria, Sep 2004. Springer-Verlag GmbH.
8. M. Broy, B. Jonsson, J.P. Katoen, M. Leucker, and A. Pretschner, editors. *Model-based Testing of Reactive Systems: Advanced Lectures*, volume 3472 of *Lecture Notes in Computer Science*. Springer, 2005.
9. E. Christensen et al. Web Service Definition Language (WSDL) ver. 1.1. http://www.w3.org/TR/wsdl/, March 2001.
10. L. Clement et al. Universal Description Discovery & Integration (UDDI) ver. 3.0. http://uddi.org/pubs/uddi_v3.htm, October 2004.
11. H.E. Eriksson et al. *UML 2 Toolkit*. John Wiley and Sons, 2004.
12. K. Bellinger et al. WS-I - basic profile, ver. 1.1. http://www.ws-i.org/Profiles/BasicProfile-1.1-2004-08-24.html, August 2004.
13. H. Foster et al. Model-based verification of web services compositions. In *Proc. ASE2003*, pages 152–161, Oct., 6-10 2003. Montreal, Canada.
14. L. Frantzen, J. Tretmans, and T.A.C. Willemse. Test generation based on symbolic specifications. In J. Grabowski and B. Nielsen, editors, *FATES 2004*, number 3395 in LNCS, pages 1–15. Springer-Verlag, 2005.

---

[3] This is for instance currently pursued within the EU Project TELCERT in the e-Learning domain.

15. X. Fu, T. Bultan, and J. Su. Analysis of interacting BPEL web services. In *Proc. of WWW2004*, May, 17-22 2004. New York, New York, USA.

16. David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch or why it's hard to build system out of existing parts. In *Proceedings 17th International Conference on Software Enginneering*, pages 179–185, April 1995.

17. H.G. Gross, I. Schieferdecker, and G. Din. *Testing Commercial-off-the-Shelf Components and Systems*, chapter Modeling and Implementation of Built-In Contract Tests. Springer Verlag, 2005.

18. M. Gudgin et al. Simple Object Access Protocol (SOAP) ver. 1.2. http://www.w3.org/TR/soap12/, June 2003.

19. R. Heckel and M. Lohman. Towards contract-based testing of web services. In *Proc. TACOS*, pages 145–156, 2004. Electr. Notes Theor. Comput. Sci. 116.

20. R. Heckel and L. Mariani. Automatic conformance testing of web services. In *Proc. FASE*, Edinburgh, Scotland, Apr., 2-10 2005.

21. C. Jard and T. Jéron. TGV: theory, principles and algorithms. In *IDPT '02*, Pasadena, California, USA, June 2002. Society for Design and Process Science.

22. B. Meyer. Applying design by contract. *IEEE Computer*, 25(10):40–51, October 1992.

23. Object Management Group. *UML 2.0 Superstructure Specification*, ptc/03-08-02 edition. Adopted Specification.

24. A. Orso, M. J. Harrold, and D. Rosenblum. Component metadata for software engineering tasks. In *Proc. EDO 2000*, LNCS 1999, pages 129–144, 2000.

25. A. Polini and A. Bertolino. *Testing Commercial-off-the-Shelf Components and Systems*, chapter A User-Oriented Framework for Component Deployment Testing. Springer Verlag, 2005.

26. J. Tretmans. Test Generation with Inputs, Outputs and Repetitive Quiescence. *Software – Concepts and Tools*, 17(3):103–120, 1996.

27. W. T. Tsai et al. Scenario-based web service testing with distributed agents. *IEICE Transaction on Information and System*, E86-D(10):2130–2144, 2003.

28. W.T. Tsai et al. Verification of web services using an enhanced UDDI server. In *Proc. of WORDS 2003*, pages 131–138, Jan., 15-17 2003. Guadalajara, Mexico.

29. XML Metadata Interchange (XMI) Specification ver. 2.0. http://www.omg.org/docs/formal/03-05-02.pdf, May 2003.