Model-Based Testing in Cloud Brokerage Scenarios

Mariam Kiran[1], Andreas Friesen[2], Anthony J.H. Simons[1], and Wolfgang K.R. Schwach[2]

[1] Department of Computer Science, University of Sheffield, Regent Court, 211 Portobello, Sheffield S1 4DP, United Kingdom
{M.Kiran,A.J.Simons}@sheffield.ac.uk
[2] SAP AG, Vincenz-Prießnitz-Str. 1, 76131 Karlsruhe, Germany
{Andreas.Friesen,Wolfgang.Karl.Rainer.Schwach}@sap.com

Abstract. In future Cloud ecosystems, brokers will mediate between service providers and consumers, playing an increased role in quality assurance, checking services for functional compliance to agreed standards, among other aspects. To date, most Software-as-a-Service (SaaS) testing has been performed manually, requiring duplicated effort at the development, certification and deployment stages of the service lifecycle. This paper presents a strategy for achieving automated testing for certification and re-certification of SaaS applications, based on the adoption of simple state-based and functional specifications. High-level test suites are generated from specifications, by algorithms that provide the necessary and sufficient coverage. The high-level tests must be grounded for each implementation technology, whether SOAP, REST or rich-client. Two examples of grounding are presented, one into SOAP for a traditional web service and the other into Selenium for a SAP HANA rich-client application. The results demonstrate good test coverage. Further work is required to fully automate the grounding.

Keywords: Model-based Testing, Cloud Service Brokerage, Cloud Broker, Web Service Testing, Lifecycle Governance.

1 Introduction

Business are shifting to Cloud computing as a new paradigm and a 5th utility service after water, electricity, gas and telephony [1] to save money on infrastructure maintenance and technical personnel. Increasingly complex Cloud ecosystems are arising, which offer various kinds of intermediation services that cater to the large number of consumers and service providers. Examples of such intermediation include finding services needed from a range of providers or marketplaces, integrating services with ERP systems, aggregating services for added-value, or monitoring and managing them. *Cloud brokerage* is the term given to explain this business model [2].

Cloud brokerage caters to a variety of capabilities supporting the needs of consumers and providers. In addition to integration and discovery, quality assurance (QA) is an important role for the broker as well. Mechanisms for QA may include such techniques as SLA monitoring, policy checks or service testing. Few examples

A.R. Lomuscio et al. (Eds.): ICSOC 2013 Workshops, LNCS 8377, pp. 192–208, 2014.
© Springer International Publishing Switzerland 2014

of such mechanisms have appeared in the Cloud so far, although CloudKick [3] provided monitoring, and Rightscale [4] provided load-balancing as services. This paper reports on some work conducted by the EU FP7 project *Broker@Cloud* that explicitly targets the functional testing of services in the Cloud, as part of a suite of quality assurance mechanisms. The paper presents a complete model-based testing methodology supporting automatic test generation for software services that are offered in Cloud brokerage scenarios.

In the rest of this paper, section 2 introduces the Cloud brokerage scenarios in which model-based testing is an enabling technology for functional QA. Section 3 presents the specification and test generation methodology. Section 4 illustrates two case studies, for which model-based tests were generated. Section 5 concludes with an analysis of the approach so far.

2 Functional Testing in Cloud Brokerage Scenarios

Previous research on service testing has come out of strategies for testing Service-Oriented Architectures (SOA) [5, 6, 7, 8]. The emphasis is on provider-based testing of services, using translations into agreed web standards [5]. For example, Bertolino et al. [6] translate category-partition testing to XML [6], Heckel et al. devise a graph-based approach [7] and a contract-based approach [8] to exercise service functional protocols in a black-box way. A few approaches [9, 10, 11] have developed finite state-based testing methods, recognizing the state-based nature of services, but find it necessary to augment web standards, which only describe service interfaces (WSDL[1]) and message formats (SOAP[2]), with additional semantic information, in order to capture how the services should behave. These research prototypes have yet to be taken up in industry, where provider-based service testing typically relies on writing manual tests to cover common usage scenarios.

In the future, functional testing may form a much stronger integral part of service development, certification and composition in Cloud ecosystems. Not only is there a need for a standard way to specify services for assuring compatibility, but testing will form part of the trust-building process at multiple stages in the service lifecycle:

- Providers will wish to offer comparable services that conform to agreed standards (in a competitive market).
- Brokers will publish these standards and offer a certification process for validating services as part of their onboarding onto a given platform.
- Consumers will want to verify their correct behaviour, before they use services, or compose applications around them.

We expect this to emerge in the same way as standards for certifying security, or for general software development. To provide such a level of assurance, it will be necessary to reduce the difficulty and cost of repeatedly recertifying services, where these are constantly evolving and being upgraded. Model-based testing is one enabling technology that may be exploited to support automatic test re-generation and

[1] Web Services Description Language.
[2] Simple Object Access Protocol.

re-testing when functional specifications are changed. Below, we describe the future business context for certifying services in the Cloud and investigate the potential benefits of model-based testing.

2.1 Cloud Brokerage and the Service Lifecycle Model

Service intermediation, or brokerage, is becoming increasingly recognized as a key component of the Cloud computing value chain [2]. We propose a Service Lifecycle Model (SLM) to describe systematically the relevant processes governing services in the context of Cloud brokerage. The SLM consists of four phases. The first three are related to the stages of service provision: *Service Engineering, Service Onboarding,* and *Service Operation*. The fourth is the on-going *Service Evolution* phase.

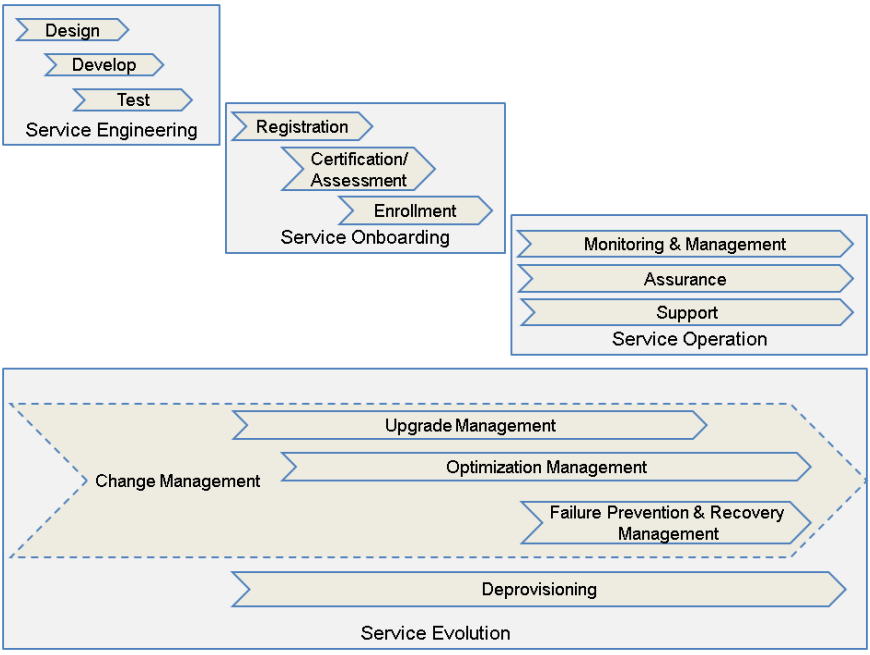

Fig. 1. The proposed Service Lifecycle Model

By analogy with software engineering, the service lifecycle starts with *Service Engineering*. The *Service Engineering* phase consists of *Design*, *Development* and *Testing* steps, carried out by the service provider. Once a Cloud service has been successfully developed and tested, and a "go to market" decision has been taken by the provider, the service enters the *Service Onboarding* phase. This phase consists of *Registration*, *Certification/Assessment* and, once the service is successfully qualified, *Enrolment*, to make the service visible to potential consumers and make it available for subscription. A service enters the *Service Operation* phase with the first consumer deciding to use the service. The most typical tasks are *Service Management* and

Assurance to manage relationships and meet agreed usage conditions. Finally, there is a fourth, *Service Evolution* phase which cuts across the whole lifespan of a service. The main task here is *Change Management*. Ultimately, the service lifespan ends with the *Deprovisioning* of the service.

It is clear that functional testing forms part of this lifecycle. Service testing currently relies on informal usage scenarios offered by the provider for certification purposes, which typically describe only part of the service's behaviour (SAP; CAS Software; SingularLogic)[3]. Testing determines whether the scenarios execute as specified, but tests are usually incomplete. While providers make use of test execution engines such as JUnit (for code) and Selenium (for web interfaces), tests are devised manually and this represents a large effort, duplicated at different stages of the lifecycle. Furthermore, there is no unified testing approach adopted by providers and brokers, since there is no shared formal specification of the service, so it is unclear whether the same QA has been applied across different service implementations, or across different host platforms. The need for commonality in service description and repeatable quality assurance after testing on multiple platforms is what distinguishes the current research from other work on service testing [5-11].

To address this, we propose a common model-based testing approach, offered as a service to providers, hosting platforms, brokers and consumers, as a means to close this interoperability gap and offer a shared level of QA. Brokers will publish common specifications and providers will agree to develop services up to these specifications. During the development stage, providers will test the services thoroughly. During the certification stage, brokers will validate the service up to the expected specification, using model-based testing. Testing may be repeated whenever a service is deployed to a different platform requiring a different implementation strategy (or *grounding*, see below), whether as a RESTful[4] web service, or a SOA-based web service using WSDL and SOAP, or even a rich-client application written in bespoke JavaScript. Internal improvements which do not change the interface may be validated by retesting. Service upgrades will need to offer a modified specification, from which all-new tests are generated. This will significantly help with the re-certification of comparable services, in a rapidly evolving Cloud ecosystem.

2.2 Model-Based Testing as an Enabling Technology

Model-based testing (MBT) is a methodology in which the designer supplies a specification, or model, that succinctly describes the behaviour of a software system, from which tests are eventually generated. The kinds of model or specification may include: a state-based specification, a functional specification, UML with OCL[5] pre- and postconditions, or a language grammar [12, 13]. The model serves as an oracle when generating tests for the system, linking specific test inputs with expected outputs [14, 5] deriving the correct results for the tests. The test generation algorithm

[3] Personal communication, Broker@Cloud industry partners.
[4] Representational State Transfer - standard HTTP running on TCP/IP.
[5] Object Constraint Language, part of the Unified Modelling Language.

also makes use of the model to determine the necessary and sufficient test coverage, up to some assumptions about the system-under-test [14].

Algorithm-driven test generation creates test-cases missed by developers (blind spots in their perceived behaviour of the system) and avoids duplicate tests that redundantly check a property more than once. The tests are then executed on the system, whose actual outputs are validated against the expected outputs. The advantages of MBT are the creation of a model, which can be internally verified for completeness and consistency, the automatic generation of test suites, the ability to determine the necessary system coverage and the automatic execution of the tests. The disadvantages are that the approach demands certain skills of testers in understanding the models, and that testing sometimes leads to state-space explosion [15].

The demands of software testing require that you drive an implementation through all of its states and transitions and observe that the implementation corresponds to the specification after each step. One of the first extended finite-state machine models to support this was Laycock's Stream X-machine (SXM) [16], which captures the behaviour of a software system in a fully-observable step-wise fashion. This work was extended by Holcombe and Ipate [14], who resolved the problem of the state explosion by abstraction into hierarchies of nested SXMs, which could be tested separately. Their proof of the equivalence of the nested machines to the expanded flat machine resulted in a tractable testing methodology that was guaranteed to find all faults in a system after testing [17].

Most work on testing software services has to date focused on Service-Oriented Architectures (SOA) rather than specifically on the Cloud [5], although the mechanisms are similar. SOA services are published using WSDL[6] interfaces that typically support testing only single operations. However, Ramollari et al. [9] presented an approach that leveraged extra semantic information attached to SAWSDL[7], in the form of production rules (RIF-PRD[8]), which supported the inference of a Stream X-Machine that was then used to generate complete functional tests. Ma et al. [10] also adopted Stream X-Machine based testing techniques to automatically generate test cases for BPEL[9] processes. Ramollari [11] used a similar approach to test SOA using explicit X-Machine specifications attached to SAWSDL service descriptions and using SOAP[10] communication. This work was the first to explore the symmetrical problems of *grounding* and *lifting*, the two-way translation between high-level abstract tests and low-level concrete tests for particular architectures.

Recent work [18, 19] has explored test generation for rich-client applications, where the application's state is maintained as a DOM[11]-tree, manipulated both by client-side user-interactions and via asynchronous AJAX callbacks from the server-side. These approaches rely on automatic inference of a state-based model of the application, from which suitable test sequences might be determined. Whereas one method [19] failed to use the model to determine full coverage, relying instead on

[6] Web Service Description Language.
[7] Semantic Annotations for WSDL and XML Schema.
[8] Rule Interchange Format-Production Rule Dialect.
[9] Business Process Execution Language.
[10] Simple Object Access Protocol.
[11] Document Object Model.

property-based testing, the other [18] converted all sequences into Selenium tests to drive the user interface, a useful approach to grounding tests for rich-client applications that we also explore in section 4 below.

3 Testing Methodology in Cloud Brokerage

We expect functional testing to be embedded into the relevant processes of the *Service Lifecycle Model*. Figure 2 illustrates the stages in the testing process. A specification (model) of a Cloud service is created and linked into its service description. This description is first published to a broker during the onboarding of the first service of its kind. Once a specification is available, high-level test sequences may be generated, offering a guaranteed level of state and transition coverage, linking expected inputs and outputs. Since these will be expressed in a platform-neutral way, it is necessary to translate the high-level tests into concrete tests, for a particular architecture, a process we call *grounding*. The concrete test suite may then be executed, to produce pass/fail test reports.

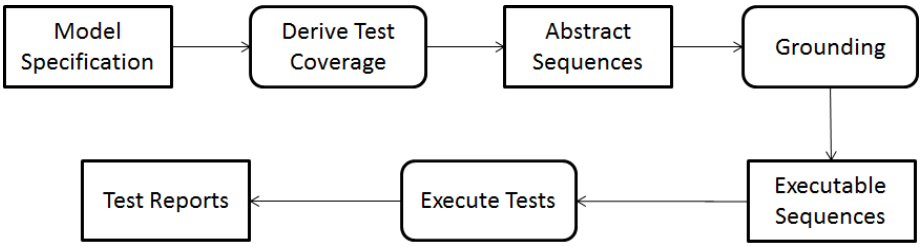

Fig. 2. Activity diagram illustrating the model-based testing methodology

This testing process may be offered at different stages of the service lifecycle. For example, a provider may use an existing service specification to generate tests for a new, replacement service in development; or a broker may perform functional testing prior to certifying a service for a particular platform. The functional testing capability may also be offered as-a-service, for the convenience of other service providers, or to consumers wishing to gain confidence in the service.

3.1 Design of the XML Specification Model

XML was chosen for the design of the common specification model, since both Cloud and SOA already make extensive use of XML. The specification of a service consists firstly of a functional part, which expresses the signatures of the service's operations and their inputs, outputs, branching conditions and state update effects on variables defined in memory. The second part consists of a finite-state machine specification, capturing the high-level control states of the service and its allowed transitions, where these are labelled with the names of distinct request/response (event/action) pairs taken from the operations. The specification language reported here is still a working prototype and is subject to revision.

The BNF for the main XML elements of the specification language is presented in figure 3, in which the notation *x ::= <y, z, …>* denotes a sequence of dependent children and *x ::= y | z |* denotes a set of alternative specializations. The set of XML attributes associated with each node are shown in set-braces.

```
Service{name} ::= <Memory, Protocol, Machine>
Memory ::= <Constant*, Variable*, Assignment*>
Protocol ::= <Operation+>
Operation{name} ::= <Request, Response+>
Message{name, type} ::= Request | Response
Request{name, type} ::= <Input*>
Response{name, type} ::= <Condition?, Output*, Effect?>
Machine{name} ::= <State+, Transition*>
State{name, initial?, final?} ::= <Transition*>
Transition{name, source, target}
Condition ::= <Predicate>
Predicate ::= Comparison | Proposition | Membership
Effect ::= <Assignment+>
Expression{name, type} ::= Parameter | Function
Parameter{name, type} ::= Constant | Variable | Input | Output
Function{name, type} ::= Assignment | Predicate | Arithmetic
  | Manipulation
Assignment{name, type} ::= <(Variable | Output), Expression>
Proposition{name, type} ::= <Predicate, Predicate>
Comparison{name, type} ::= <Expression, Expression>
Membership{name, type} ::= <Expression, Expression>
Arithmetic{name, type} ::= <Expression, Expression>
Manipulation{name, type} ::= <Expression, Expression, Expression?>
```

Fig. 3. BNF (Backus-Naur Form) of the service specification language

3.2 Procedure for Generating Complete Functional Tests

A version of the Stream X-Machine test generation algorithm [10, 7] was used to generate high-level test sequences from the specification. The algorithm determines the state cover by breadth-first search, then constructs languages of events, consisting of all possible interleaved sequences of length 1, 2, .., *k*, up to some chosen coverage criterion. These are concatenated onto the state cover to generate the high-level coverage sequences. For low values of *k = 2..4*, it is possible to ensure that:

- all specification states exist in the implementation;
- no unexpected states exist, such as ill-behaved clones of the expected states;
- all specified target states of transitions also exist in the implementation;
- no unexpected transitions exist in the implementation.

The sequences were then simulated in a model of the machine and protocol, to determine which sequences should be accepted or rejected. Attempting to traverse a missing transition should always be rejected, whereas traversing a present transition may be allowed conditionally, according to the guards governing each response. Where guards govern an input, more than one test case should be generated, to cover each input partition. The result is a tree of high-level tests, also expressed in XML, corresponding to positive sequences that should succeed, and negative sequences that should fail, when presented to the implementation.

The automatic algorithm ensured that every distinct case in the specification was covered by at least one test; and also that the tests were minimal (non-redundant) and exhaustive up to the assumptions in the specification. The algorithm determined the extent of testing needed to achieve the coverage goals, up to assumptions about redundancy in the implementation. The algorithmic nature of test generation means that it is possible to re-test, or generate new tests (after a service upgrade) to the same coverage levels, promoting a degree of uniformity in QA.

4 Analysis and Evaluation via Case Studies

Two case studies were developed to prototype the test grounding strategy. The first study was a traditional web service, implemented using Java, WSDL and SOAP. The second study was a rich-client application developed for the SAP HANA Platform-as-a-Service (PaaS), which currently offers independent software vendors (ISVs) a platform and a manual certification process for onboarding their third-party web services. Whereas the first study focused on the feasibility of translating high-level tests into SOAP, the second study also investigated ways of supplying grounding information for creating Selenium tests, as an additional part of the specification.

4.1 Case Study: A Shopping Cart Web Service

The first case study was created as a stand-alone web service, as though developed by a provider seeking to offer a SOA application, similar to others available in the Cloud. The provider was allowed to develop the service as they liked (c.f. the *Service Engineering* phase), and also provided the specification for it, indicating the service's expected behaviour using the XML specification language of figure 3.

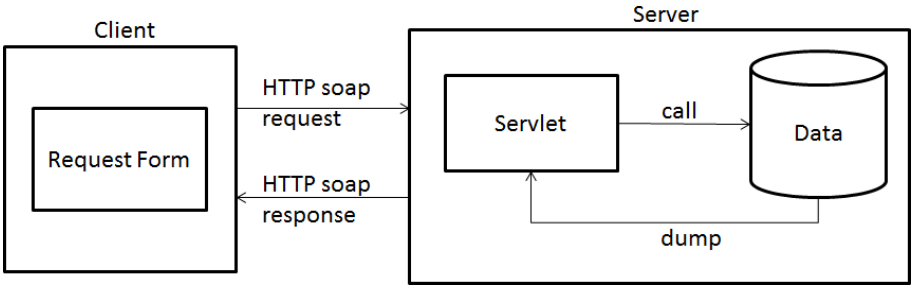

Fig. 4. Client-server architecture for a *Shopping Cart* web service

Figure 4 illustrates the client/server architecture of the *Shopping Cart* web service. The client's presentation logic offered a list of items to purchase, as shown in the screenshot of figure 5. A Java servlet modelled the control logic, whose high-level control states and transitions are illustrated by the diagram in figure 6.

Fig. 5. The web form offered by the client-side of the *Shopping Cart*

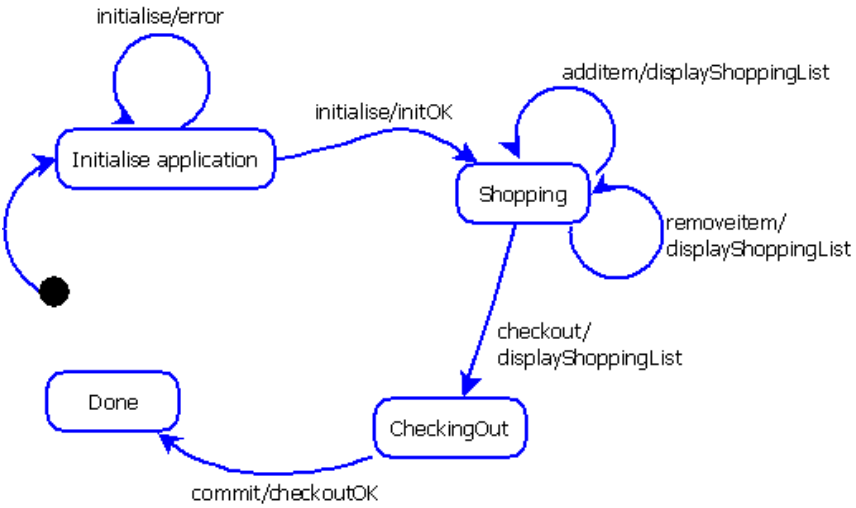

Fig. 6. The state machine for the server-side of the *Shopping Cart*

The memory-state of the application, which corresponded to the items currently ticked in the web-form, was stored in a database on the server-side. Communication

between the client and server was via SOAP messages and WSDL interfaces. The client-side issued commands as SOAP requests, which were interpreted on the server-side. After responding and updating the data state, the server returned a SOAP response, indicating the action taken. The client-side used Java wrappers to issue SOAP messages and interpret the responses, which allowed comparison of actual and expected values.

Figure 6 shows the state machine representing the intended design of the *Shopping Cart* service. The states: *S = {InitialiseApplication, Shopping, CheckingOut, Done}* represent the control stages in the shopping lifecycle. The alphabet of the machine: A = {*initialise/error, initialize/initOK, additem/displayShoppingList, removeitem/ displayShoppingList, checkout/displayShoppingList, commit/checkoutOK}* represents the complete set of operations. The state-transition logic shows when particular operations are allowed, for example, the client must connect successfully to the shopping service before adding items to the cart; and after checking out no further items may be added. The labels on the transitions correspond to event/action pairs that eventually correspond to SOAP request/response messages, in this implementation.

The state machine and abstract functional behaviour were encoded in the XML specification language, and then passed to an independent agent acting as a broker. From this, abstract coverage sequences were generated by algorithm, using the parameter setting *k=1*, yielding the transition cover (sufficient for an implementation with no redundant states). This test-set attempts to reach every single state and then fire every possible valid and invalid transition. The sequences were filtered by the machine to identify present/missing transitions, and then filtered by the functional specification, to identify the conditions guarding certain transitions.

```
<TestSuite id="0">
  <Sequence id="1" source="Init" target="Init"/>
  <Sequence id="2" source="Init" target="Shopping">
    <TestStep id="3" name="initialise/initOK">
      <Request id="4" name="initialise" type="Request">
        <Condition id="5">
          <Comparison id="6" name="equals" type="Boolean">
            <Variable id="7" name="isServerReady" type="Boolean"/>
            <Constant id="8" name="true" type="Boolean"/>
          </Comparison>
        </Condition>
      </Request>
      <Response id="9" name="initOK" type="Success"/>
   </TestStep>
 </Sequence>
  …  <!-- omitted further Sequence elements -->
</TestSuite>
```

Fig. 7. Fragment of a high-level test suite for the *Shopping Cart*

Figure 7 illustrates a fragment of the resulting high-level test suite. The first empty sequence denotes a test to determine whether the application can be initialized. The second sequence shows the single step necessary to reach the *Shopping* state, with extra information about the memory-state of the server needed to satisfy the precondition guard. The response denotes a *positive* test step confirming expected behaviour; *negative* test steps were also synthesized for requests that should be ignored in certain states (corresponding to missing transitions in the model).

These high-level tests were given back to the provider, who then had the task of grounding these sequences as SOAP request/response pairs, where a request transmitted the input data, and a response showed which action had been triggered. In this first study, the grounding was performed by hand, following simple rules for converting the high-level tests. The purpose of this was to determine whether a mapping to SOAP was feasible, and whether testing could observe the properties required by the testing method [14, 16, 17]. For example, to achieve *output distinguishability*, it is necessary to identify uniquely which transition fires in response to each input.

```
<?xml version="1.0" encoding="UTF-8"?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Header/>
  <S:Body>
    <ns2:getSelectedProduct xmlns:ns2="http://service.amazonian.org/">
      <pnames>tele</pnames>
    </ns2:getSelectedProduct>
  </S:Body>
</S:Envelope>
<?xml version="1.0" encoding="UTF-8"?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Body>
    <ns2:getSelectedProductResponse
         xmlns:ns2="http://service.amazonian.org/">
      <return> * Good News, You have just bought: tele</return>
    </ns2:getSelectedProductResponse>
  </S:Body>
</S:Envelope>
```

Fig. 8. SOAP request and response for the *addItem/displayShoppingList* action

Figure 8 shows the SOAP request sent when the user checks the *tele* box on the client-side. The response indicates the success of the *addItem* request. Further SOAP responses were created for each action, including planned error-handling and an explicit null response for events that are ignored in the current state, corresponding to missing transitions in the state-transition diagram. A JUnit test driver was built on the client-side, driving a Java EE wrapper-class that issued the SOAP requests and unpacked the SOAP responses. During testing, it was found that the implemented service did not always signal explicitly when it had ignored a request, as required by the specification. This was considered a successful testing outcome.

4.2 Case Study: A SAP HANA Cloud Application

The second case study was created as a software service, designed to be deployed on an existing Cloud platform, SAP HANA. As above, the provider was allowed to develop the service, but designed this up to a specification, written in the XML specification language, that was regulated by an independent agent, acting as a broker. Unlike the previous case study, which used the open standards WSDL and SOAP, this study had a bespoke rich-client implementation, so would prove significantly more difficult to test, in the grounding phase.

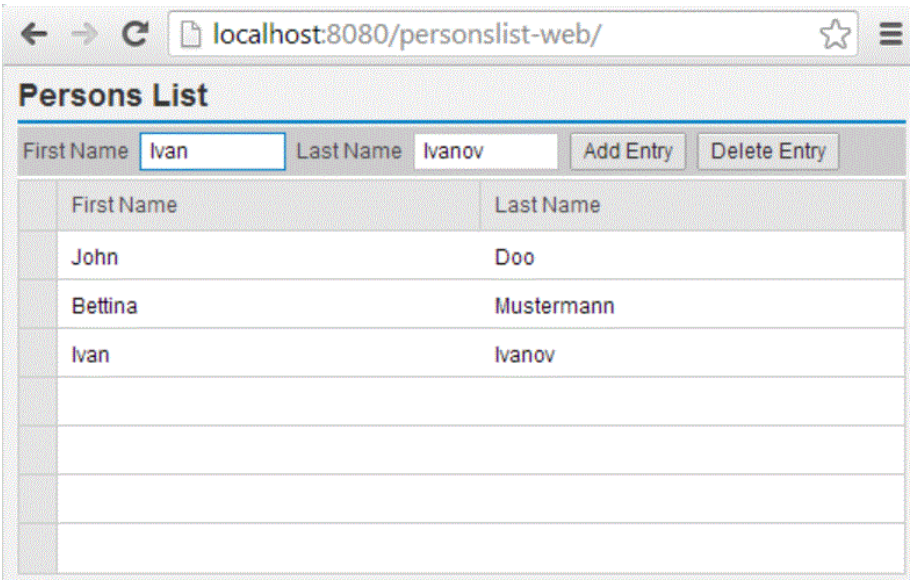

Fig. 9. Rich-client application for a *Contact List* built on the SAP HANA platform

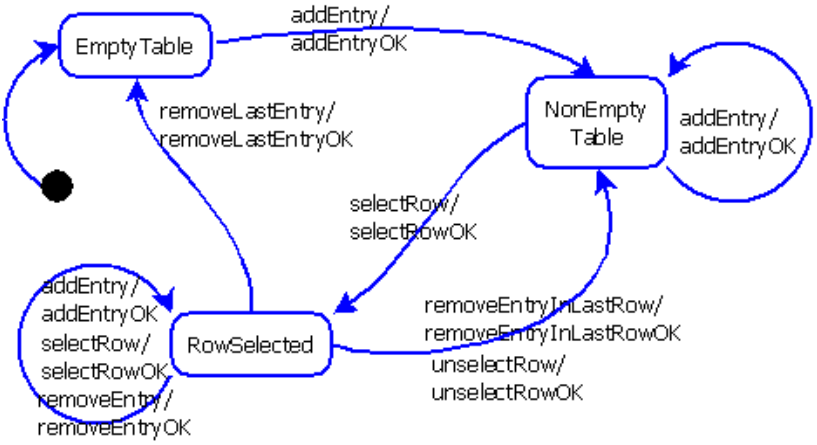

Fig. 10. The state machine for the SAP HANA *Contact List* application

Figure 9 shows a screenshot of the web service for maintaining a *Contact List*, deployed as a rich-client application on the SAP HANA Cloud PaaS. The application allows the user to add names, or remove selected names from a list of contacts. Figure 10 shows the corresponding control logic of the state machine specification. Only the successful cases of each operation were modelled explicitly, using guards on the responses. Thus, all errors were treated as missing transitions, which were later signalled using a pop-up dialog box in the grounding of the tests. In the case of failure, the state should not change. High-level test coverage sequences were generated from the model, in the same manner as described above.

In this second case study, we were particularly interested in providing broker support for testing a non-standard or unknown implementation. In such a situation, it is clearly the provider's responsibility to identify the route to grounding the high-level test suite. The provider decided that the best strategy was to treat the rich-client's web interface as the entry-point to the service, so suggested grounding the tests as generated Selenium code, to drive the client's browser in predetermined ways.

Selenium [20] is a tool that is conventionally used for recording user interactions in a web-browser, which can later be replayed as tests. Here, the provider wished to derive the concrete Selenium tests from the high-level test sequences, derived in turn from the broker's specification (see figure 2) and use them to drive client's browser, and hence the deployed SAP HANA Cloud service, through all of its states and transitions. The goal was to convert all abstract sequences into concrete Selenium tests, which, if executed without reporting errors, signify that the application has conformed to the specification.

```
<TestSuite id="0">
   <Sequence id="1" source="EmptyTable" target="EmptyTable"/>
   <Sequence id="2" source="EmptyTable" target="NonEmptyTable">
      <TestStep id="3" name="addEntry/addEntryOK">
         <Request id="4" name="addEntry">
            <Input id="5" name="forename" type="String"/>
            <Input id="6" name="surname" type="String"/>
         </Request>
         <Response id="7" name="addEntryOK" type="success"/>
      </TestStep>
   </Sequence>
  …  <!-- omitted further Sequence elements -->
</TestSuite>
```

Fig. 11. Fragment of generated high-level test suite for the *Contact List*

Figure 11 focuses on the abstract test sequence with *id=2*, showing the inputs required by the *addEntry* request that triggers the *addEntryOK* response. To support the grounding to Selenium, extra grounding information was added to the functional part of the specification of the *addEntry* operation, shown in figure 12.

```
<Operation name="addEntry">
   <Request name="addEntry">
      <Input name = "forename" type = "String">
         <Grounding>
            <Target>Selenium<Target/>
            <ElementType>TextField</ElementType>
            <ElementID>firstNameFieldId</ElementID>
            <TestValue>John</TestValue>
         </Grounding>
      </Input>
      <Input name="surname" type="String">
         <Grounding>
            <Target>Selenium<Target/>
            <ElementType>TextField</ElementType>
            <ElementID>lastNameFieldId</ElementID>
            <TestValue>Smith</TestValue>
         </Grounding>
      </Input>
      <Grounding>
         <Target>Selenium<Target/>
         <ElementType>Button</ElementType>
         <ElementID>addPersonButtonId</ElementID>
         <Action>click</Action>
      </Grounding>
   </Request>
   <Response name="addEntryOK" type="success">
      <Grounding>
         … <!-- omitted grounding info, to report success  -->
      </Grounding>
      … <!-- omitted Effect, for updating memory -->
   </Response>
</Operation>
```

Fig. 12. Fragment of the functional specification for *Contact List*, with grounding information

The new idea here is that an XML sub-language for grounding may be created to support the grounding of the high-level test suites in any particular technology. The Selenium engine is driven by a table of instructions provided in an XML DOM-tree[12], which also records the before- and after-states of each interaction. In figure 12, the *Grounding* nodes contain extra information about the Selenium DOM-tree elements to insert as part of the test input data, along with the Selenium button-click event that should be triggered to fire the *addEntry* request. This information was created by the service provider, in a similar format to the original functional specification. The bespoke grounding algorithm ensured that, whenever an *addEntry* request was listed

[12] Document Object Model - memory representation of an XML file.

in the high-level tests (see figure 11), this would be matched against the *addEntry* operation in the grounding information (see figure 12), which supplied the input and button-click data for generating the concrete Selenium test instructions. Code snippets from the DOM-tree generated for the Selenium driver are shown in figure 13:

```
<tr> <td>type</td> <td>id=firstNameFieldId</td> <td></td> </tr>
<tr> <td>sendKeys</td> <td>id=firstNameFieldId</td> <td>John</td> </tr>

<tr> <td>type</td> <td>id=lastNameFieldId</td> <td></td> </tr>
<tr> <td>sendKeys</td> <td>id=lastNameFieldId</td> <td>Smith</td> </tr>

<tr> <td>click</td> <td>id=addPersonButtonId</td> <td></td> </tr>
```

Fig. 13. Sample code generated for the Selenium test driver, for the *Contact List*

As above, the grounding algorithm was only partially automated; but enough was learned to see how a fully automated method might be developed, using translation strategies similar to the *Visitor Design Pattern* [21]. It seems likely that, whereas a broker may eventually be expected to provide standard groundings for SOAP and WSDL services, non-standard implementations will always require bespoke grounding strategies, supplied by the service provider.

During testing of this application, it was found that the implemented service did not exactly follow the state-based model, as required by the specification. After removing the last entry from the table, it was found that the implemented service remained in the state *RowSelected* instead of switching into the state *EmptyTable*. This was a successful testing outcome demonstrating the discovery of a (non-obvious) incorrect behavior.

5 Conclusions

This paper presents early results from the development of a standard method and a supporting mechanism for automated functional testing in the Cloud. The mechanism supports (at least) the certification phase of a Service Lifecycle Model, as operated by Cloud service brokers, and may also support providers during the service engineering phase, and consumers during the operation phase. Some service consumers may also be providers, seeking to compose larger services out of smaller ones, hence will be interested in validating component services in the Cloud.

Central to this effort is the development of a common service specification language. The XML specification language was able to model adequately the two case studies described, and is also fairly close in its syntax to other service description languages, such as Linked USDL[13] [22], so is likely to be acceptable in the community. Once a specification has been parsed, the resulting model also supports symbolic checking for the completeness of the specification (for missing transitions and exhaustiveness of the guards). This is essential if the mechanism is to be widely

[13] Unified Service Description Language.

adopted by developers who are not necessarily trained in formal methods. The fully automatic generation of high-level tests was successful in achieving levels of coverage not yet found in manual service testing in industry. This was borne out in the feedback from industry partners (SAP; CAS Software; SingularLogic) and also demonstrated in the detection of some non-obvious faults in the case studies.

The work on automated grounding is still incomplete, but a manually-assisted grounding strategy was shown, for the sake of demonstrating the general strategy and the fault-finding potential of the concrete tests. Future work will concentrate on building an improved model simulator and test oracle; and on developing automatic groundings for certain standard service implementation technologies. This may go some way towards the goal of providing Testing-as-a-Service in the Cloud [23].

Acknowledgment. The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 328392, the Broker@Cloud project (www.broker-cloud.eu).

References

1. Buyya, R., Yeo, C.S., Venugopal, S., Broberg, J., Brandic, I.: Cloud Computing and Emerging IT Platforms: Vision, Hype, and Reality for Delivering Computing as the 5th Utility. Future Generation Computer Systems 25, 599–616 (2008)
2. Plummer, D.C., Lheureux, B.J., Karamouzis, F.: Defining Cloud Services Brokerage: Taking Intermediation to the Next Level. Report ID G00206187. Gartner, Inc. (2010)
3. Rao, L.: Using CloudKick to manage Amazon Webservices' EC2. TechCrunch, http://techcrunch.com/2009/03/16/y-combinators-cloudkick-offers-simple-cloud-management-system/ (March 16, 2009)
4. Higginbotham, S.: Rightscale Makes Multiple Clouds Work. GigaOM, http://gigaom.com/2008/09/17/rightscale-makes-multiple-clouds-work/ (September 17, 2008)
5. Bozkurt, M., Harman, M., Hassoun, Y.: Testing & Verification in Service-Oriented Architecture: A Survey. Software Testing, Verification and Reliability 32(4), 261–313 (2012)
6. Bertolino, A., Frantzen, L., Polini, A., Tretmans, J.: Audition of Web Services for Testing Conformance to Open Specified Protocols. In: Reussner, R., Stafford, J.A., Szyperski, C. (eds.) Architecting Systems. LNCS, vol. 3938, pp. 1–25. Springer, Heidelberg (2006)
7. Heckel, R., Mariani, L.: Automatic Conformance Testing of Web Services. In: Cerioli, M. (ed.) FASE 2005. LNCS, vol. 3442, pp. 34–48. Springer, Heidelberg (2005)
8. Heckel, R., Lohmann, M.: Towards Contract-based Testing of Web Services. In: Proc. Int. Workshop on Test and Analysis of Component Based Systems, Barcelona, Spain. ENTCS, vol. 116, pp. 145–156 (2004)
9. Ramollari, E., Kourtesis, D., Dranidis, D., Simons, A.J.H.: Leveraging Semantic Web Service Descriptions for Validation by Automated Functional Testing. In: Aroyo, L., et al. (eds.) ESWC 2009. LNCS, vol. 5554, pp. 593–607. Springer, Heidelberg (2009)
10. Ma, C., Wu, J., Zhang, T., Zhang, Y., Cai, X.: Testing BPEL with Stream X-Machine. In: Proceedings of the 2008 International Symposium on Information Science and Engineering, pp. 578–582. IEEE Computer Society, Shanghai (2008)

11. Ramollari, E.: Automated Verification and Testing of Third-Party Web Services. PhD Thesis, Dept. of Computer Science, University of Sheffield, UK (2012)
12. Utting, M., Legeard, B.: Practical Model-Based Testing: A Tools Approach. Morgan Kaufmann, Burlington (2007)
13. Pretschner, A., Philipps, J.: Methodological Issues in Model-Based Testing. In: Broy, M., Jonsson, B., Katoen, J.-P., Leucker, M., Pretschner, A. (eds.) Model-Based Testing of Reactive Systems. LNCS, vol. 3472, pp. 281–291. Springer, Heidelberg (2005)
14. Holcombe, W.M.L., Ipate, F.: Correct Systems - Building a Business Process Solution. Applied Computing Series. Springer, Berlin (1998)
15. El-Far, I.K., Whittaker, J.A.: Model-Based Software Testing. In: Marciniak, J.J. (ed.) Encyclopedia of Software Engineering. John Wiley & Sons, London (2002)
16. Laycock, G.: The Theory and Practice of Specification Based Software Testing. PhD Thesis. Dept. of Computer Science, University of Sheffield, UK (1993)
17. Ipate, F., Holcombe, W.M.L.: An integration testing method which is proved to find all faults. Int. J. Comp. Math. 63, 159–178 (1997)
18. Marchetto, A., Tonella, P., Ricca, F.: State-Based Testing of Ajax Web Applications. In: Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation, pp. 121–130. IEEE Computer Society Press, Washington, DC (2008)
19. Mesbah, A., van Deursen, A., Roest, D.: Invariant-Based Automatic Testing of Modern Web Applications. IEEE Trans. Software. Eng. 38(1), 35–53 (2012)
20. Selenium, H.Q.: Browser Automation, `http://www.seleniumhq.org/`
21. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns - Elements of Reusable Object-Oriented Software. Addison-Wesley (1996)
22. KMI and SAP Research: Linked USDL, `http://www.linked-usdl.org`
23. Yang, Y., Onita, C., Dhaliwal, J., Zhang, X.: TESTQUAL: conceptualizing software testing as a service. In: Proc. 15th Americas Conf. on Information Systems, USA, paper 608 (2009)