Model-Based Testing for Composite Web Services in Cloud Brokerage Scenarios

Mariam Kiran[1], Anthony J. H. Simons[2]

[1] School of Electrical Engineering and Computer Science, University of Bradford, Bradford, BD7 4DP, UK
`m.kiran@bradford.ac.uk`
[2] Department of Computer Science, University of Sheffield, Regent Court, 211 Portobello, Sheffield, S1 4DP, UK
`a.j.simons@sheffield.ac.uk`

Abstract. Cloud brokerage is an enabling technology allowing various services to be merged together for providing optimum quality of service for the end-users. Within this collection of composed services, testing is a challenging task which brokers have to take on to ensure quality of service. Most Software-as-a-Service (SaaS) testing has focused on high-level test generation from the functional specification of individual services, with little research into how to achieve sufficient test coverage of composite services. This paper explores the use of model-based testing to achieve testing of composite services, when two individual web services are tested and combined. Two example web services – a login service and a simple shopping service – are combined to give a more realistic shopping cart service. This paper focuses on the test coverage required for testing the component services individually and their composition. The paper highlights the problems of service composition testing, requiring a reworking of the combined specification and regeneration of the tests, rather than a simple composition of the test suites; and concludes by arguing that more work needs to be done in this area.

1 Introduction

Cloud computing is becoming a prevalent business paradigm for software delivery and services, allowing businesses to save on the costs of infrastructure, maintenance and personnel [1]. To enable this, complex cloud computing environments are emerging that support new business models for cloud service and infrastructure providers, to help manage this increase in demand. Among the various cloud scenarios such as private, public and multi-cloud scenarios, cloud brokerage is one which is quickly becoming popular and difficult to manage.

Cloud brokerage is still an active research area, bringing challenges of risk, security and trust [2, 3] and further issues in terms of how brokers handle services, recommend optimal infrastructures and perform cloud service quality checking when they link customers to cloud environments [4]. By acting as an intermediary between the service consumers and providers, the brokers are expected to ensure that all requirements of the services are met and delivered on time.

Given the need for mechanisms to assure the quality of service for risk and security, testing of the services is another challenging and expensive task, with brokers scrutinizing infrastructures and applications over issues of reliability, functionality and performance. Cloud services consist of using service-oriented architecture applications which focus on Software-as-a-Service (SaaS) functionality. A large body of literature exists focusing on how various functional attributes of services can be tested using approaches like fault-based testing, model-based testing and interoperability testing [11]. However, these approaches have focused on specific individual services being used independently.

This paper contributes to the area of service testing by focusing on the principles of functional testing for composed cloud services. The paper describes how the specification of the composite service needs to be reworked and the service tested again, even after the individual services have passed all test cases. This is due to issues of interoperability and integration of the components in the new composite service, which interact in ways not anticipated in the original component specifications. A model-based testing tool [30] is used to demonstrate systematically the kind of test coverage required to achieve the same levels of quality assurance for simple and composed software services. The tool attempts to automate the testing procedure as much as possible.

The paper has been organized in the following manner: Section 1 presents an introduction to cloud brokerage and the testing challenges for individual web services. Section 2 discusses the related work in this domain, discussing cloud environments and the test research used to produce test suites for services. Section 3 describes the model based testing approach, with examples of two web services: the login and the shopping services. Section 4 discusses the issues when the two services are composed to produce a composite shopping cart service. Finally Section 5 and 6 present the problems encountered, leading to the conclusions of this paper and the future work in this domain.

2 Related Work

2.1 Testing in Cloud Brokerage Ecosystems

Cloud computing adopts three broad styles of software architecture, when communicating between nodes. These are as follows:

- HTTP requests and responses, known as Representational State Transfer (REST). This is a "lightweight" approach, where the client is a simple web-browser and data is transferred in compact HTTP formats; but it requires bespoke server-side processing to dispatch requests.
- Service-Oriented Architecture (SOA) adopts XML standards, using SOAP for message communication, WSDL and UDDI for service description and discovery. SOA technology supports open, extensible, federated and composable architecture and fosters the separate development of autonomous,

modular software components, which can be reconfigured in various ways before usage [5]. In this respect, SOA is vendor-diverse, offering the prospect of reusable, interoperable web-services [6], also offering a means of describing and measuring the Quality-of-Service (QoS) arising out of the distributed nature of services [7, 8].

- An increasingly popular style uses bespoke rich-client desktops, providing app-like services that use continuous information trickle via AJAX to communicate with back-end servers. Rich-client applications are developed in client-side scripting languages, such as JavaScript, resulting in *thick client* MVC applications. This architecture presents a different set of testing challenges [9, 10] and like RESTful services, does not lead to homogeneity.

Much research has been conducted into developing tools to test SOA, which arguably may also apply to the cloud [11]. However, clouds are more challenging due to their heterogeneous nature, involving many different kinds of stakeholders, integrating many packages that operate asynchronously. Cloud brokers are faced with merging services of more than one of the above kinds, to assess the trustworthiness of composite applications desired by consumers. This involves assessing and certifying complex service oriented applications which are composed of distributed software services that can be selected dynamically, assembled together and executed to produce evolving software ecosystems.

2.2 Functional Testing Approaches for Composite Web Services

Web services use open standards and are quite flexible to accommodate fault tolerance, security or performance requirements [12]. 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 and provider-based testing of services, using translations from agreed web standards [7]. Further work [29] has used labelled transition systems to define the testing of web services, based on their protocols. While web services may be used individually, accessed through simple HTTP or SMTP protocols, a more interesting prospect is when they are combined in more complex applications.

Figure 1 describes a typical service-oriented architecture, in which communication takes place between three actors: the service provider, the service requester (also known as the consumer) and the service broker. The service provider publishes (1) a service interface (WSDL) to the broker's UDDI registry. The service requester then contacts the UDDI registry to discover (2) a suitable service and find out who the provider is; and then the broker acts as intermediary (3) in closing the deal. The service requester thereafter communicates with the service provider directly (4) using the SOAP message protocol. Since all SOAP data is transmitted as XML, the service provider may validate service requests; likewise the requester may validate the response from the provider, using a suitable XSD file (XML Schema).

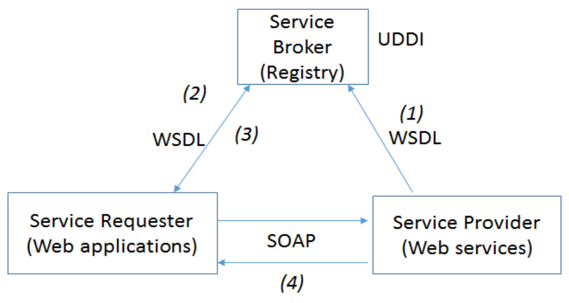
Service Broker (Registry)
UDDI
(2)
WSDL
(3)
(1)
WSDL
Service Requester (Web applications)
SOAP
Service Provider (Web services)
(4)

Fig. 1. Interaction between provider, requester and broker in SOA c.f. [31].

In cloud computing environments, the providers are responsible for providing the necessary SOA infrastructure middleware and infrastructural mechanisms for service discovery, discovery to service providers, consumers and integrators. Broker can act as a service integrator use existing services to create composite services to create an end user application. In such cases, brokers are responsible for developing guidelines for testing composites of various types. This involves employing a range of composition mechanisms (e.g. WSBPEL, application-embedded, WS-CDL).

Testing composed services involves a workflow management to make interoperability possible by focusing on the interfaces for data transfer. Researchers have used Mealy models for defining complex state based operations in services [14] or data driven approaches to OWL-S [15, 16], however these fail to describe how different test suites for services can be merged if individual services are tested.

3 Towards a Methodology for Composite Service Testing for Cloud Brokerage

3.1 Testing During the Cloud Service Lifecycle

Kiran et al. [17] have described the use of model-based testing for cloud brokerage scenarios. Model-based testing depends on some kind of model specification, either a state-based specification, or a functional specification, or a combination, such as UML with OCL[1] pre- and post-conditions; essentially using any modelling formalism with a formal language grammar [18, 19]. The model serves as an oracle when generating tests for the system, linking specific test inputs with expected outputs [20, 11], deriving the correct results for the tests. The test generation algorithm also uses the model to determine the necessary and sufficient test coverage, given some reasonable assumptions about the system-under-test [20].

[1] Object Constraint Language, part of the Unified Modelling Language

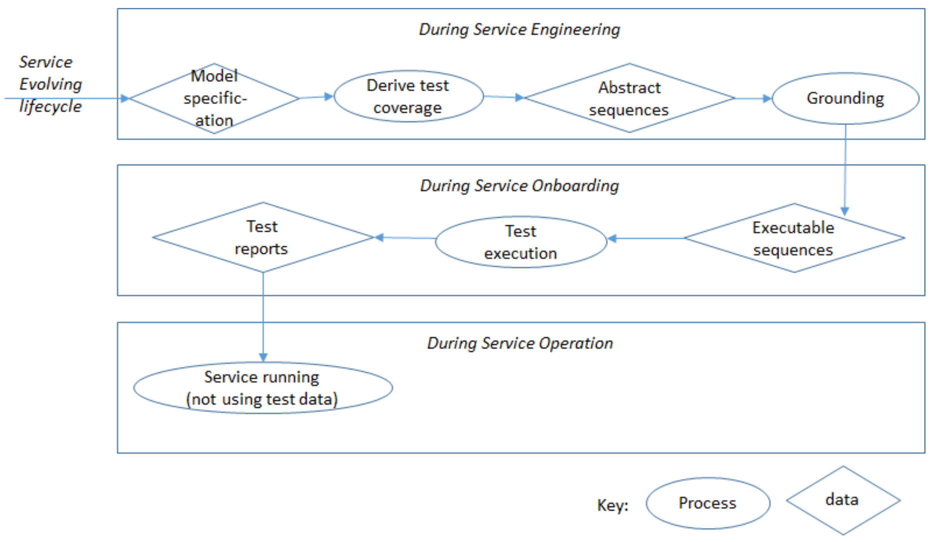

Fig. 2. Service evolution through the lifecycle when services are engineered, tested during onboarding, and when deployed to execute on the infrastructure.

Figure 2 illustrates how broker-managed testing might be organised during the onboarding phase of the service lifecycle. The diagram shows the development stages for service *specification*, which describes the service formally, *test generation*, which produces abstract test sequences directly from the specification, and *grounding*, which translates the high-level tests into concrete tests capable of execution on a given service architecture [17], after which the concrete test suite may be used to test a service implementation to produce pass/fail test reports. Testing during the service onboarding phase is likely to be an important part of service certification by brokers. Testing will still be carried out as usual by service providers during the service engineering phase; and potential service consumers may also wish to re-test the services that they include in service compositions, for added quality assurance.

3.2 Testing Considerations for Composed Web Services

The following general quality assurance considerations apply to platforms offering service discovery, conformance testing, and service composition [13, 24], where test case generation, execution and verdict assignment can be focused according to the service lifecycle stage:

1. Testing service discovery, as part of the Service Engineering phase
   a. Define what properties should be described
   b. Define how to query against them efficiently
2. Testing service composition, as part of the Service Engineering phase
   a. Specify the goals of a composition
   b. Specify the constraints on a composition
   c. Build a composition from component services

    d. Analyse the composition
3. Testing data flows, during the Service Onboarding phase
    a. How to keep initial data separated
    b. How to track data movement between services
    c. How to provide the transactional guarantees

During composite service testing, component services are treated as pieces to be glued together. This gives rise to the naïve idea that composed test-sets might be derived cheaply by considering how the services are combined. Services can be combined in three ways: (1) *sequentially*, by ordering services one after the other - this is equivalent to joining two services on their respective final and initial states; (2) *concurrently*, when the combined services are executing together in parallel - this is not tractable, since every action of one service could interleave at any point with the actions of the other; and (3) *decision-based,* a variant of sequential combination, where the path to follow depends on a condition [25] (Figure 3).

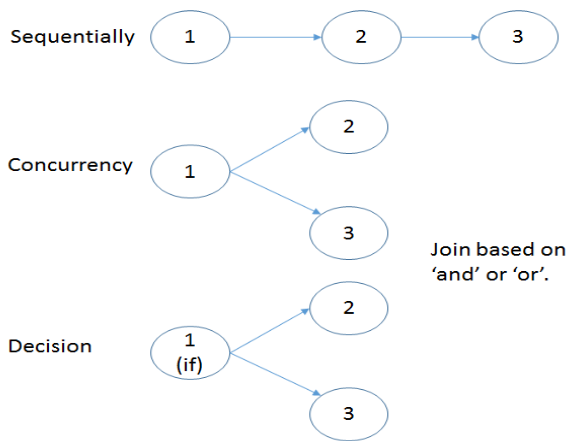

Fig. 3. Sequential, concurrent and decision arrangement when composing services.

Other researchers have used various approaches for composite testing using model checking for temporal logic using finite state machines [21] or using rule-based services [22]. Enkatraman et al. [23] showed a dynamic verification of protocol compliance in commitments modelled using auction behaviour. Tsai et al. [26] used rank and fault detection to find the capability of test scripts to establish an oracle for most test inputs to test the complete composite service. Endo et al. [27] employed an event-driven model approach to support the test model and generation, to test the environment and also support test concretization (grounding) and test execution.

4 Generating Model-based Test Suites for Composite Services

The eventual goal of composite service testing is to make it easier for the average developer to produce high-quality tests for composed services, possibly on-the-fly at run-time. In the naïve approach, tests are created and archived for each component service, but are combined by rule at a later stage. The example in fig. 4 presumes that a *Login* service and simple *Shopping* service are to be composed as an authenticated *ShoppingCart* service. Decision-based composition initially seems to be appropriate, under the condition specified in table 1:

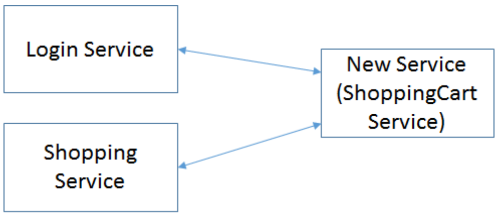

Fig. 4. The *Login* and *Shopping* service being combined for the *ShoppingCart* service.

Table 1. Decision Table for *ShoppingCart* service.

	Rules	
Constraints	R1	R2 (opposite of R1)
Server is Ready	X	¬X
Next Action		
Login service		X
Shopping Service	X	

The decision table shows that if R1 is true (*server is ready*) then the *Shopping* initial state is reached, else the *Login* final state is reached. Such a decision table can help build composed services, based on the constraint rules, by conditionally joining their state transition graphs. The testing procedure could be something like:

1. Generate test cases.
    a. Generate and store all paths for the *Login* service.
    b. Generate and store all paths for the *Shopping* service.
2. Combine test cases.
    a. Generate a decision table for the combined *ShoppingCart* service.
    b. Create joined test sequences by combining paths.
    c. Create concrete test inputs for the combined paths.

We will show that this approach is inadequate, because of a faulty intuition about sequential (and hence decision-based) composition. A different model of composition is required to express faithfully what happens when services are joined.

4.1 Login and Shopping Services Modelled Separately

First we introduce the FSM concepts briefly to show how the test suites for each web service are generated. FSMs consist of a finite number of states, one of which is an initial state and the rest are intermediate and final states. To model the execution of a web service, every transition in a FSM corresponds to a web request/response cycle. Figure 5 show the FSMs for two component services, the *Login* service and the simple *Shopping* service. These figures are the visual representation of the specifications, developed in a model-based specification language [30], which defines all states, transitions, service requests and responses received. Based on these specifications, the model-based testing tool creates test suites for both the *Login* and the *Shopping* service.

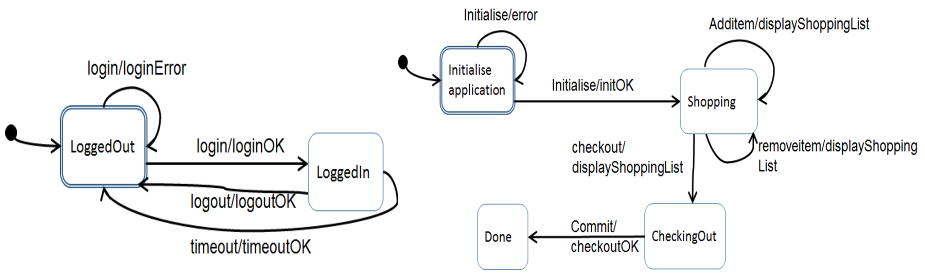

Fig. 5. FSM for *Login* Service and *Shopping* Service

```
<TestSuite id="0" testDepth="1">
 <Notice id="1" text="Generated test suite for service: LoginService">
  <Analysis id="2" text="Exploring all paths up to length: 1"/>
  <Analysis id="3" text="Number of theoretical sequences: 9"/>
  <Analysis id="4" text="Number of infeasible sequences: 0"/>
  <Analysis id="5" text="Number of executable sequences: 9"/>
 </Notice>
 <TestSequence id="6" state="LoggedOut" path="0">
  <TestStep id="7" name="create/ok" state="LoggedOut" verify="true">
   <Operation id="8" name="create"/>
  </TestStep>
 </TestSequence>
 <TestSequence id="9" state="LoggedOut" path="1">
  <TestStep id="10" name="create/ok" state="LoggedOut">
   <Operation id="11" name="create"/>
  </TestStep>
  <TestStep id="12" name="login/ok" state="LoggedIn" verify="true">
   <Operation id="13" name="login">
    <Input id="14" name="userName" type="String">Jane Good</Input>
    <Input id="15" name="password" type="String">serendipity</Input>
   </Operation>
  </TestStep>
 </TestSequence>

 <!-- other sequences omitted for brevity -->
</TestSuite>
```

Fig. 6. Fragment of a test suite generated for the *Login* Service. Some data from the above XML has been omitted for reasons of brevity.

Figure 6 shows a fragment of the transition cover test set generated for the *Login* service. This reaches every state, and explores every single transition from each state. This fragment shows just the first two sequences from this test set, which represent the initial (empty) sequence that should reach the *LoggedOut* state; and a valid login sequence that should reach the *LoggedIn* state. The tool generates all realizable positive test cases (that should be present) and negative test cases (that should not be present). The output is prefixed by metadata describing the possible number of theoretical sequences (in the state machine), which may sometimes be pruned, but in this example are all realizable (the guards permit all the transitions). The tester may choose the maximum path length; typically a value slightly greater than 1 is chosen, since the implementation may not be a minimal state machine

4.2 Composing the Two Services as a ShoppingCart Service

When these two services are merged together to produce a composed service, they are *not* actually joined in any linear fashion on their initial and final states, but rather, the entire behaviour of the *Shopping* state machine is embedded inside the *LoggedIn* state of the *Login* machine. The correct composition model for this is to use nested state machines, known as *Compound FSMs* or *Hierarchical FSMs* [28].

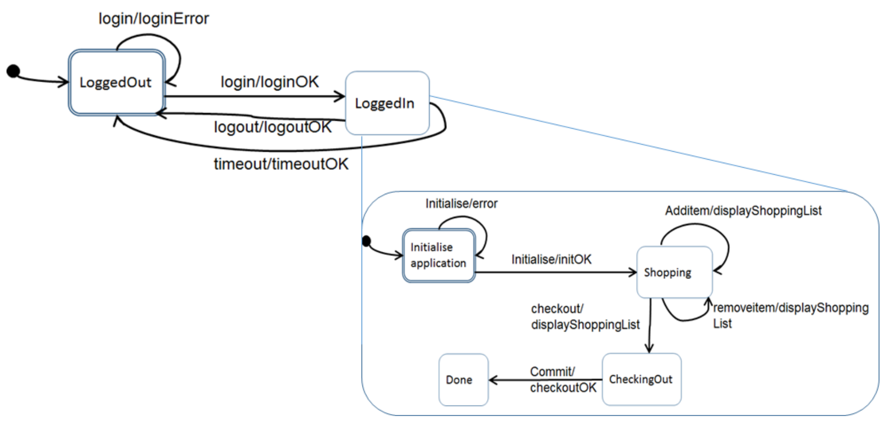

Fig. 7. *ShoppingCart* Service FSM (Composite service of *Login* and *Shopping* Services)

If we adopt the semantics from Harel and UML state machines, then any transition entering the *LoggedIn* superstate is deemed to enter the initial substate of the embedded *Shopping* machine; and any transition exiting this superstate is deemed to exit every contained substate. It is then possible to flatten this hierarchical FSM, to remove the superstate and create an equivalent flat state machine. According to Ipate [28], there are two possible strategies for generating tests for hierarchical FSMs:

- Flatten the hierarchical model and generate tests for the flattened machine – this is adequate, but produces much larger test sets, as a result of the transition explosion resulting from longer paths to reach all states;
- Treat the state hierarchy as a kind of refinement, and develop separate test suites for the external and internal FSMs, finding some way to integrate the expanded sequences for paths that traverse a superstate boundary.

The first approach was applied to the model in figure 7, yielding the flattened model in figure 8. The *login/loginOK* transition to the *LoggedIn* superstate was replaced by a transition from *LoggedOut* to the *InitialiseApplication* initial state of the embedded *Shopping* service. All of the *Shopping* service's substates were then given exit transitions for every exit transition leaving the *LoggedIn* superstate in the *Login* machine. They acquired additional transitions *timeout/timeoutok* and *logout/logoutok* as shown in Figure 8.

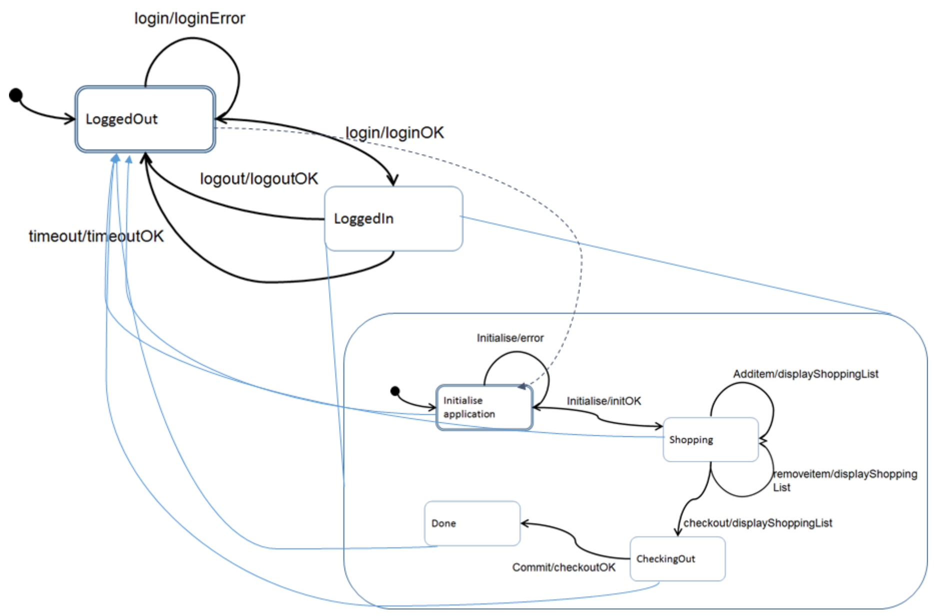

Fig. 8. Flattening the *ShoppingCart* Service FSM. For brevity, the substate exit transitions for timeout/timeoutOK and logout/logoutOK are each represented by single arrow.

5 Experimental Test Generation Results

The objective of this research was to investigate the two testing methods described by Ipate [28], using our testing tool for software services [30]. Initially, we expected the flattening approach to be computationally expensive, resulting from an exponential growth in test suite size. The hierarchical FSM modelling approach then might point the way towards a smarter refinement testing strategy.

In practice, the testing tool reduces the number of theoretical sequences generated for flattened examples, by pruning tests that either cannot be executed, or which replicate the results of other tests. Initially the tool generates all positive paths (which must exist) and all negative paths (which must not). However, once a negative transition has been proven absent, then all longer sequences containing this as a prefix may be pruned (assuming that memory is unchanged when an event is ignored).

Similarly, the set of all positive paths includes some unrealizable sequences (blocked by guards on the current input and memory); so these are impossible, even in the specification, and are also pruned.

Figure 9 shows the metadata generated by a longer test of the *Login* service, where the maximum path length has been increased to 2 (anticipating one redundant state per desired state in the implementation). The count of infeasible sequences includes all pruned sequences; in this case, paths containing negative prefixes (such as a repeated login attempt when already logged in) have been pruned, after each negative test has been satisfied once.

```
<TestSuite id="0" testDepth="2">
 <Notice id="1" text="Generated test suite for service: LoginService">
  <Analysis id="2" text="Exploring all paths up to length: 2"/>
  <Analysis id="3" text="Number of theoretical sequences: 37"/>
  <Analysis id="4" text="Number of infeasible sequences: 16"/>
  <Analysis id="5" text="Number of executable sequences: 21"/>
 </Notice>
...
```

Fig. 9. Test Suite metadata generated for Login service, with path length 2.

Test suites were generated for the individual component *Login* and *Shopping* services, and also for the composed *ShoppingCart* service. The metadata statistics for these are shown in table 2. From this, it is clear that while the theoretical size of the composed test set for the flattened state machine increases exponentially, the practical test set generated is a lot smaller than this, as a result of test pruning. Nonetheless, the resulting test set is not a simple combination of the test sets for the two composed services.

Table 2. Statistics on test sequences for *Login*, *Shopping* and *ShoppingCart* services using statistics generated by the tool in figure 9.

	LoginService (depth =1)	*LoginService (depth=2)*	*Shopping Service (depth =2)*	*Shopping Service (depth= 3)*	*Shopping Cart Service (depth =2)*	*Shopping Cart Service (depth=3)*
Theoretical Sequences	9	37	151	907	511	5111
Infeasible sequences	0	16	120	846	372	4735
Executable sequences	9	21	31	61	139	376

The testing tool reported that a path length of 3 was eventually necessary to cover all the transitions of the *ShoppingCart* service. This is because operations depended on particular values of memory variables, such as an item necessarily being present in the cart, before proceeding to checkout. Because of the longer paths explored through the combined machine, we expected a steep rise in the number of test sequences generated. The theoretical test suite size grows exponentially with longer paths; an upper bound[2] may be estimated from the recurrence relation: $C*\Sigma\alpha^k$, where C is the size of the state cover, α is the size of the event alphabet, and k increments from zero to the maximum path length. Nonetheless, table 2 shows that in some cases (as with this example), it might be tractable to compose state-based specifications by embedding and then generate tests from the equivalent flattened specification; after all, a test suite consisting of 376 tests sequences is not actually terrible, particularly if test generation and test execution are automated.

Looking at the degree of pruning in the original component services, it is clear that the greatest reduction in test suite size was contributed by the simple *Shopping* service, for which the tool pruned 93% of the theoretical test paths (at *depth* = 3), which were found to be either redundant or non-realizable. This is intuitively due to the "staged" nature of a shopping application, where many operations are disabled in particular states; and this only needs to be proved once for each operation. By contrast, fewer paths were pruned for the *Login* service. For the composite service, 92% of paths were pruned (at *depth* = 3), which is highly useful.

Considering the theoretical test explosion in table 2, it is attractive to speculate whether it might be possible to generate test suites for composite services by composing (in a more principled fashion) the test suites generated for the component services. It is clear that this will be no simple pooling of the component test suites; one would need to generate additional "glue sequences" to verify that the two machines were correctly joined together. Ipate previously speculated on the idea of refining the paths of the outer machine, by splicing in all paths through the nested machine, when the outer paths traverse the relevant superstate boundary [28]. In the conclusions below, we suggest an approach in which certain simplifying assumptions about the composed services allow you to identify sets of "glue sequences".

6 Conclusions and Future work

This paper set out to determine whether it was tractable to develop test suites for composite software services either by reusing the test suites generated for the component services, or by reusing the component specifications in some way. From the theory of testing FSM-based specifications, we expected to find that there was no easy way of reusing the component test suites to achieve the same level of coverage of the composite service. However, we found that it is possible to compose and flatten state machine specifications, and from this, regenerate all-new tests for the composite service, to the same level of coverage. The test suites for the composite service turned out to be more tractable than anticipated, due to the test path pruning behaviour of the testing tool [30], which eliminates redundant paths with null-op transitions in the prefix, and unrealizable paths for which tests cannot be executed.

To achieve any better reduction than this requires making quite strong assumptions about the services being composed. The most important assumption is that the *sets of events processed by each service do not intersect*; this allows consideration of the behaviour of each service in isolation. Without this assumption, when testing the

[2] Some sequences computed by the recurrence relation already exist in the state cover; the actual test suite is a set and contains no duplicate sequences.

composite service, the events of both FSMs must be pooled and many more negative tests are required, to demonstrate a lack of mutual interference between the FSMs. However, if the non-intersection assumption holds, then it is feasible to consider an approach where a composite test suite consisting of the component test suites, plus some additional “glue sequences”, might be thought satisfactory. For the composed *ShoppingCart* service, the glue sequences would have to ensure that:

- Every transition of the *Login* service entering the *LoggedIn* state also enters (or enters instead) the initial state of the *Shopping* service.
- Every transition of the *Login* service exiting the *LoggedIn* state also exits every state in the *Shopping* service and targets the *LoggedOut* state.

Depending on whether the *LoggedIn* state is preserved, or expanded away in the composition, the first glue sequence may be considered additional, or a replacement for one of the *Login* service’s sequences. The remaining glue sequences must reach the state cover of the inner nested FSM, and then exercise the “glue transitions” leading back to the outer machine. This “extended state cover” is easily constructed by prefixing the state cover of the nested *Shopping* machine by the sequence from the *Login* machine that reaches the *LoggedIn* superstate. Altogether, in this example, there would be nine “glue sequences”: one path to verify that the *login/loginOK* entry transition reaches the *Shopping* initial state; and two paths for each *Shopping* state, to verify that the *logout/logoutOK* and *timeout/timeoutOK* exit transitions lead back to the *LoggedOut* state.

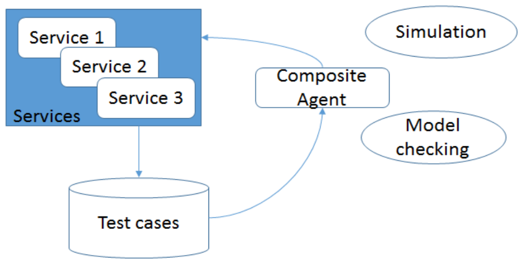

Fig. 10. Using composite agents to test composite web services.

Figure 10 sketches one possible architecture for testing composite services. The idea is based around a composition tool, here called a *Composite Agent*, that is able to reason in the manner described above, when composing FSM specifications. The agent would have access to the database of service specifications (the specifications include a high-level FSM describing control states, plus a more detailed description of the service’s operations and their effects on memory [30]), and from this, would be able to calculate the additional “glue sequences” required, when services were composed by embedding one service’s FSM inside a state of the other FSM. These “glue sequences” could be returned, on demand, along with the available test-suites for testing each of the individual component services.

The advantages of such an approach includes the fact that “glue sequences” could be generated on-the-fly, as services were composed dynamically. This would be very useful for creating "late integration" test suites, assuming that the components had already been tested. Secondly, the approach is very flexible: in principle, any service could be embedded inside any state of any other service; and multiple services could be composed, by embedding each component service in a different superstate of the composite service. This approach is also fully compositional, in that FSMs could be nested to arbitrary depths.

The drawbacks of this approach include the strong assumption that must be made about the non-interference of FSMs. This is partly mitigated by the fact that the *Composite Agent* may check that the alphabets of each composed machine do not intersect (assuring the separation of machines, in principle). As described above, the "glue transitions" do not anticipate any redundancy in the implementation. They only verify every single-step test obligation needed to show that the composed services appear to be properly connected. However, it would also be possible to generate slightly longer sequences that guarantee this with a higher level of confidence (c.f. testing redundant EFSM implementations with slightly longer paths [20]).

Compound services are a complex and interesting proposition for testing. Further work needs to explore how full automated test suites can be generated from individual test suites plus "glue sequences". The ideal solution should make use of component specifications, which may be composed on-the-fly. Such a capability would be of great advantage to cloud service brokers.

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 (http://www.broker-cloud.eu).

References

1. R. Buyya, C.S Yeo, S. Venugopal, J. Broberg, I. Brandic, Cloud Computing and Emerging IT Platforms: Vision, Hype, and Reality for Delivering Computing as 5th utility. Future Generation Computer Systems, 25, 599 – 616, 2008.
2. A.U. Khan, M. Kiran, M. Oriol, M. Jiang and K. Djemame: Security risks and their management in cloud computing. CloudCom pp: 121-128 2012.
3. M. Kiran, M. Jiang, D. Armstrong, K. Djemame, Towards a Service Life Cycle-based Methodology for Risk Assessment in Cloud Computing, Cloud and Green Computing, 2011.
4. D.C. Plummer, B.J. Lheureux, and F. Karamouzis, Defining Cloud Services Brokerage: Taking Intermediation to the Next Level. Report ID G00206187, Gartner, Inc., 2010.
5. M. Bell, Introduction to Service-Oriented Modeling. Service-Oriented Modeling: Service Analysis, Design, and Architecture. Wiley & Sons. p. 3. ISBN 978-0-470-14111-3, 2008.
6. Arcitura Education Inc, Service Orientation, Online: http://serviceorientation.com/, 2012.
7. D. Norton, J. Feiman, N. McDonald, M. Pezzini, Y. Natis, D. Sholler, G. Heiden, F. Karamouzis, A. Young, G.A. James, E. Knipp, J. Duggan, T. Murphy, R. Valdes, M. Blechar, M. Driver, G. Young, J. Vining, R. Knox, D. Feinberg, T. Hart, C. Patrick, J. Forsman, M. Basso, R. Simpson, Y. Adachi, W. Clark, M. King, J. Hill, D. Gootzit, A. Bradley, L. Kenney, D. Stang, Hype Cycle for Application Development, Gartner, 2009.

8. H. Mei, L. Zhang, A framework for testing web services and its supporting tool Proceedings of IEEE Int. on Service-Oriented System Engineering. Computer Society, 2005, 199–206.
9. A. Marchetto, P. Tonella, F. Ricca., State-Based Testing of Ajax Web Applications. Proc. of the Int. Conf. on Software Testing, Verification and Validation. IEEE Computer Society, 121-130, DOI: 10.1109/ICST.2008.22, 2008.
10. A. Mesbah, D. Roest, Invariant-Based Automatic Testing of Modern Web Applications, January/February 2012, vol. 38, no. 1, pp. 35-53.
11. M. Bozkurt, M. Harman, and Y. Hassoun, Testing & Verification in Service-Oriented Architecture: A Survey Software Testing Verification and Reliability, 2009, 10.1002/000.
12. Web Services Architecture (W3C Working Group).
13. R. Hull, J. Su, Tools for Design of Composite Web Services, Presented Version (June 17, 2004), Online: http://www.cs.ucsb.edu/~su/tutorials/sigmod2004.html
14. G.H. Mealy, A Method to Synthesizing Sequential Circuits, Bell System Technical Journal. pp. 1045–1079, 1955.
15. M. Klusch, A. Gerber, Evaluation of Service Composition Planning with OWLS-XPlan, Online: http://www-ags.dfki.uni-sb.de/~klusch/i2s/klusch-evaluation-owlsXPlan.pdf
16. B. Norton, S. Foster, A. Hughes, A Compositional Operational Semantics for OWL-S, www.dip.deri.org/documents/Norton-et-al-A-Compositional-Semantics-for-OWL-S.pdf
17. M. Kiran, A. Friesen, A.J.H. Simons, W.K.R. Schwach, Model-based Testing in Cloud Brokerage Scenarios ICSOC, 2013.
18. M. Utting and B. Legeard, Practical Model-Based Testing: A Tools Approach. Morgan-Kaufmann, Burlington MA, 2007.
19. A. Pretschner, J. Philipps, Methodological Issues in Model-Based Testing. In: Broy, M. et al. (eds.) Model-Based Testing of Reactive Systems, LNCS, Vol. 3472, pp. 281–291, 2005.
20. W.M.L. Holcombe and F. Ipate, Correct Systems - Building a Business Process Solution. Applied Computing Series, Springer-Verlag, Berlin Heidelberg New York, 1998.
21. X. Fu, T. Bultan, and J. Su, Analysis of interacting Bpel web services, Proceedings of the 13th international conference on World Wide Web, ACM Press, pp. 621-630, 2004.
22. S. Deutsch, A. Vianu, Specification and Verification of Data Driven Web Services, 2004.
23. M.V Enkatraman, M.P Singh, Verifying Compliance with Commitment Protocols Enabling Open Web-Based Multiagent System, 2010.
24. T. Cao, P. Felix, R.Castanet, I. Berrada, Online Testing Framework for Web Services, Testing Composite Web Services.
25. F. Belli, M. Linschulte, An Event-Based Approach, April 2009.
26. W. T. Tsai, Y. Chen, R. Paul, N. Liao and H. Huang, Cooperative and Group Testing in Verification of Dynamic Composite Web Services, 2011.
27. A.T. Endo, M.B. Silveira, E. Macedo, R. Simao, F.M. de Oliveiray, A.F. Zorzo, Using models to test web service-oriented applications:an experience report, 2012.
28. F. Ipate, Test Selection for Hierarchical and Communicating Finite State Machines, The Computer Journal, Vol. 52, No. 3, 2009.
29. A. Bertolino, L. Frantzen and A. Polini, Audition of web services for testing conformance to open specified protocols, Architecting Systems with Trustworthy Components, no. 3938, LNCS, 2006.
30. A.J.H. Simons, Cloud Service Quality Control: Broker@Cloud Verification and Testing Tool Suite. http://staffwww.dcs.shef.ac.uk/people/A.Simons/broker/, 2014.
31. B. Wu, B. Zhou, L. Xi, Remote multi-robot monitoring and control system based on MMS and web services, Industrial Robot: An International Journal, vol 34, no. 3, pp 225-239, 2007.