Proceedings Paper:
Lefticaru, R. and Simons, A.J.H. (2015) X-Machine Based Testing for Cloud Services. In: Ortiz, G. and Tran, C., (eds.) Advances in Service-Oriented and Cloud Computing. Workshops of ESOCC 2014, September 2-4, 2014, Manchester, UK. Lecture Notes in Computer Science, 508. Springer, pp. 175-189. ISBN: 978-3-319-14885-4. ISSN: 1865-0929.

https://doi.org/10.1007/978-3-319-14886-1_17

X-Machine Based Testing for Cloud Services

Raluca Lefticaru[1,2] and Anthony J. H. Simons[1]

[1] Department of Computer Science, The University of Sheffield
Regent Court, Portobello Street, Sheffield S1 4DP, UK
r.lefticaru@sheffield.ac.uk, a.j.simons@sheffield.ac.uk
[2] Department of Computer Science, University of Bucharest
Str. Academiei nr. 14, 010014, Bucharest, Romania

Abstract. In this article we present a tool designed for cloud service testing, able to generate test cases from a formal specification of the service, in form of a deterministic stream X-machine (DSXM) model. The paper summarizes the theoretical foundations of X-machine based testing and illustrates the usage of the developed tool on some examples. It shows in detail how the specification should be written, which are the design for test conditions it should satisfy, in order to assure the generation of high quality test suites for the cloud service.
Keywords: cloud service testing; state-based testing; X-machine.

1 Introduction

As cloud computing has emerged as a new paradigm for hosting and delivering services over the Internet [16], the enterprise IT environment has been transformed into a matrix of interwoven infrastructure, platform and application services which are delivered from different service providers. In this context, Cloud Service Brokers will play an important role by serving as intermediaries between providers and consumers, ensuring the quality of software-based enterprise cloud services.

This paper provides a powerful model-based testing approach for cloud services, which aims to increase the confidence in the quality of service behaviour, previously tested using a specification in the form of a *stream X-machine* (SXM). The testing methodology based on SXMs is more general and can be applied to other types of software, not only cloud services, but cloud ecosystems would substantially benefit from this approach, which would allow testing and trusting all the services that agreed on implementing the same SXM specification and passed the generated test sets.

A stream X-machine [12, 11] is a particular type of X-machine [9], a state model which has been investigated for many years, because of perceived advantages in: (1) its modelling capability, e.g. has been used in high performance agent based simulators like FLAME [7]; and (2) its associated testing methods [12, 11, 8]. In essence, an SXM is a class of extended finite state machine (EFSM), having an internal memory and transitions labelled by processing functions, which might have input parameters and can update the machine internal memory. An

SXM can model not only the control part of a system, but also the data processing. And, if certain design for test conditions are satisfied, from the SXM specifications one can derive test suites able to establish the correctness of the implementation under test (IUT). Much recent research has been focused on obtaining theoretical results, such as relaxing the conditions the SXM should satisfy (determinism, nondeterminism, controllability), in order to produce high quality test suites.

In this paper we use a deterministic SXM to model the cloud service that will be tested and we present a tool, which assists the user in writing the SXM specification, validating it and generating test data.

The paper is structured as follows: Section 2 introduces the theoretical foundations of stream X-machine testing, Section 3 presents the tool developed, some examples are summarized in Section 4, related work is presented in Section 5 and finally conclusions are drawn in Section 6.

2 Theoretical Background

In this section we present the foundations of stream X machine testing. We will use ε to denote the empty sequence. For a finite alphabet Σ, Σ^* represents the set of all finite sequences with members in Σ. For $a, b \in \Sigma^*$, ab denotes the concatenation of the two sequences a and b. For $U, V \in \Sigma^*$, $UV = \{ab | a \in U, b \in V\}$; U^n is defined by $U^0 = \{\varepsilon\}$ and $U^n = U^{n1}U, n \geq 1$. Furthermore, $U[n] = \bigcup_{0 \leq i \leq n} U_i$.

2.1 Stream X-Machines

A Stream X-Machine (SXM) is an extended form of state machine, capable of modelling both the data and the control of the system [11, 12]. Compared to Finite State Machines (FSMs), SXMs enrich them with: (1) internal storage or *memory* (the internal variables of the machine), (2) *processing functions* instead of input/output symbols which traditionally labelled the transitions of FSMs.

Definition 1. *An* SXM *[12] is a tuple* $Z = (\Sigma, \Gamma, Q, M, \Phi, F, q_0, m_0)$ *where:*

- Σ *is the finite* input alphabet
- Γ *is the finite* output alphabet
- Q *is the finite* set of states
- M *is a (possibly infinite) set called* memory
- Φ *is a finite set of distinct* processing functions*; a processing function is a non-empty (partial) function of type* $M \times \Sigma \longrightarrow \Gamma \times M$
- F *is the (partial)* next-state function*,* $F : Q \times \Phi \longrightarrow Q$
- $q_0 \in Q$ *is the* initial state
- $m_0 \in M$ *is the* initial memory value

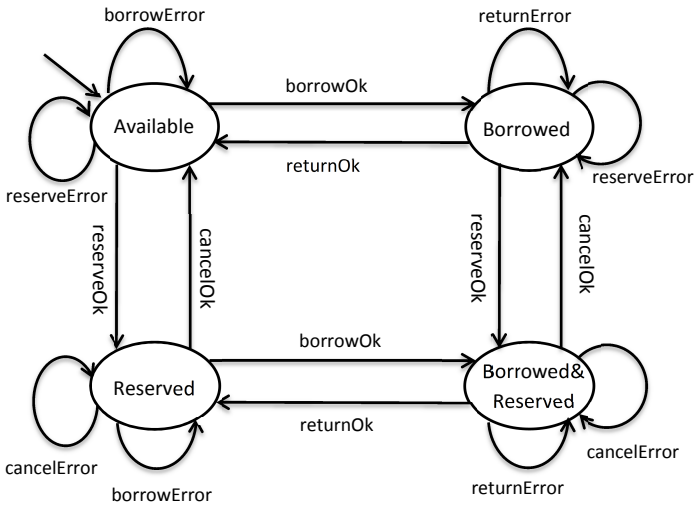

Fig. 1. A state transition diagram for an SXM modelling a Book service

When representing a certain system, it is frequently helpful to abstract and think of an SXM as a finite automaton with the arcs labelled by functions from the set Φ. The automaton $A_Z = (\Phi, Q, F, q_0)$ over the alphabet Φ is called the *associated finite automaton of* Z.

Usually, the processing functions of Φ specify the components or the possible operations of the system specified by Z. The memory typically represents the variables used by the modelled system. It is formed from tuples, where each element of the tuple corresponds to either a global variable or a parameter that may be passed between the elements of Φ.

When modelling a real system, the input alphabet Σ consists of the names of the operations which should be triggered and their parameters, if applicable. For example, a simplified book lending service, shown in Fig. 1, could have operations like $borrow(id)$, $return(id)$, $reserve(id)$, $cancel(id)$, where id is a number representing the customer which attempts to do the operation, with values taken from a finite set. However, providing an actual value for the parameters, and depending on the current state and value of the SXM memory, there might exist different processing alternatives. For example, the SXM can have two processing functions, labelled *borrowOk* and *borrowError* (in order to differentiate them as distinct branches of the operation $borrow(id)$ from the implementation). These two processing functions can be triggered if the value of the id is adequate for the first case or if the value of id is incorrect in the second case (e.g., the book is reserved by another customer, represented by the internal memory variable res_id, and $res\_id \neq id$).

There are cases in which the number of processing alternatives is more than two (success/error). For example, a bank account is modelled in [8] as a SXM having $\Sigma = \{open(), deposit(a), withdraw(a), close()\}$, where a represents a

positive number. The SXM can have different functions, such as $WithdrawAll$, $Withdraw$, $WithdrawError$ (written in upper case, in order to differentiate them from the input symbol $withdraw(a)$), which can be triggered if the value of a is equal to the account balance b in the first case, $0 < a < b$ in the second case and $b < a$ for the error case. For example, a bank account is modelled in [8] as a SXM having $\Sigma = \{open(), deposit(a), withdraw(a), close()\}$, where a represents a positive number. Even if $a > 0$ takes values from an infinite set, thus making Σ also infinite, the theoretical results concerning the X-machine test generation will still be preserved, the only difference being that the search process for appropriate input values will be realized in an infinite set.

In this context, when the SXM has more functions able to process the same input, like $borrow(id)$ or $withdraw(a)$, one desirable quality of the SXM is to be *deterministic* (abbreviated DSXM) which means that there exists at most one possible transition for any triplet (state, memory, input). This implies that the domains of the processing functions are disjoint. This property is not unusual for a cloud service, it is normal to expect a deterministic, repeatable behaviour, whenever the system is in the same configuration and receives the same input values. However, there exist also testing methodologies developed for non-deterministic X-machines [2] where the non-determinism can be caused either by overlapping functions domains or by non-determinism of the associated automaton A_Z, i.e. $F : Q \times \Phi \longrightarrow 2^Q$.

Definition 2. *An SXM Z is said to be* deterministic *if for every $\phi_1, \phi_2 \in \Phi$, if there exists $q \in Q$ such that $(q, \phi_1), (q, \phi_2) \in domF$ then either $\phi_1 = \phi_2$ or $dom\phi_1 \cap dom\phi_2 = \emptyset$;*

A sequence $p \in \Phi^*$ of processing functions induces a function $||p||$ that shows the correspondence between a (memory, input sequence) pair and the (output sequence, memory) pair produced by the application, in turn, of the processing functions in the sequence p.

Definition 3. *Given $p \in \Phi^*$, $||p|| : M \times \Sigma^* \longrightarrow \Gamma^* \times M$ is defined by:*

- $||\epsilon||(m, \epsilon) = (\epsilon, m), m \in M$
- *Given $p \in \Phi^*$ and $\phi \in \Phi$, $||p\phi||(m, s\sigma) = (g\gamma, m')$, for $m, m' \in M$, $s \in \Sigma^*$, $g \in \Gamma^*$, $\sigma \in \Sigma$, $\gamma \in \Gamma$ such that there exists $m'' \in M$ with $||p||(m, s) = (g, m'')$ and $\phi(m'', \sigma) = (\gamma, m')$.*

A computation of the SXM represents the traversal of all sequences of transitions in the associated automaton A_Z and the successive application of the corresponding processing functions (using the actual parameters provided to each of them) to the initial memory value for the first function and to the resulting memory value for the following processing functions. The correspondence between the input sequence applied to the machine and the output produced gives rise to the relation (function) computed by the Z.

Definition 4. *The relation computed by Z, $f_Z : \Sigma^* \longleftrightarrow \Gamma^*$ is defined by: $(s, g) \in f_Z$ if there exist $p \in \Phi^*$ and $m \in M$ such that $(q_0, p) \in domF^*$ and $||p||(m_0, s) = (g, m)$.*

Definition 5. *An SXM Z is said to be* completely-defined *if* $dom f_Z = \Sigma^*$.

In other words, an SXM is completely-defined if every sequence of inputs can be processed by at least one sequence of functions accepted by the associated automaton. In the case when the SXM has some "refused" (or ignored) inputs, it can be transformed into a completely-defined one by adding a designated error output, which is not in the output alphabet of Z and completing the automaton with self-looping transitions or transitions to an extra (error) state.

2.2 Reaching and Distinguishing States in a DSXM

When testing from finite state machines (FSMs) or from finite automata, some important notions are:

- *state cover*, a set S consisting of sequences that reach every state of the machine;
- *transition cover*, a set T consisting of sequences that reach every state of the machine and exercise every transition from that state; if S is a state cover and X the input alphabet, then the transition cover can be computed by $T = S \cup SX$
- *characterization set*, usually labelled W, that distinguishes between every pair of states in the FSM.

Considering the automaton from Fig. 1, where the input alphabet is $X = \{borrowOk, borrowError, returnOk, returnError, reserveOk, reserveError, cancelOk, cancelError\}$, a state cover can be $S = \{\varepsilon, borrowOk, reserveOk, borrowOk\, reserveOk\}$, and a characterization set $W = \{returnOk, cancelOk\}$.

Using these sets, test suites of the form $Y = TX[k]W$ can be produced, according to the W-method [6], where S is a state cover, $T = S \cup SX$ a transition cover, X is the input alphabet of the machine, W the characterization set and $k \geq 0$ is the difference between the estimated number of states in the implementation and the number of states of the specification. The W-method is the most general testing method (that does not rely on the existence of direct state inspection). For this method to be applicable, there must be a reliable *reset* in the implementation, that correctly puts the system specified by the FSM into its initial state before each test sequence is executed.

The idea behind the test suite $Y = TX[k]W$ is that $T = S \cup SX$ ensures that all the states and transitions in the specification are also present in implementation, the set $X[k]W$ verifies that the implementation is in the same state as the specification after triggering each transition. In case the implementation contains up to k extra states, the set $X[k]W = X^kW \cup \ldots \cup \{\varepsilon\}W$ ensures that each of them would be reached by some input sequence of length up to k and that they behave the same way as the corresponding specification states (by applying the sequences from W, which can distinguish between states).

Many adaptations of the W-method exist in the literature, such as the *round trip* approach [1], based on a transition tree constructed in a depth-first fashion, which includes all the transition sequences that begin and end with the

same state. Other authors [4] preferred to use a reliable state-reporting oracle instead of the characterization set W, in order to check the current state of the implementation under test.

The testing methods for simple FSMs have been adapted to SXM testing [10, 2, 11, 12], and consequently corresponding notions have been proposed to build the theoretical framework, such as realisable sequences, r-state cover, separating sets (which are sets of sequences of processing functions that differentiate between every pair of separable states of the machine [12]).

Because the transitions in the state diagram of an SXM are labelled not by simple input/output symbols as in FSMs, but by functions, with restrictions on their input/memory which prevent them from firing unconditionally, there might exist states that are reachable in the associated automaton, but which cannot be reached by any input sequence applied to the machine. This is why the state cover from the FSM should be replaced, when applying testing methods from SXM, with an r-state cover, which is a minimal set of realisable sequences, that reaches every r-reachable state in Z [11, 12].

Analogously, there may be pairs of distinguishable states in the associated automaton for which the sequences of processing functions that distinguish between them can never be applied.

2.3 Design for Test Conditions

The first approach on testing using X-machines, the so called "DSXM integration testing" [15, 13, 10, 2], was inspired by the W-method, and it can guarantee the conformance of the implementation, with respect to the SXM specification. The method was originally developed for testing the control structure of a system, modelled by a DSXM and it can be applied under some design for test conditions and the assumption that the processing functions of the X-machine have been correctly implemented (and previously tested).

The two design for test conditions necessary in this approach are: *output-distinguishability* and *input-completeness*.

Definition 6. *An SXM Z is said to be* output-distinguishable *if for all $\phi_1, \phi_2 \in \Phi$, whenever there exist $m, m_1, m_2 \in M, \sigma \in \Sigma, \gamma \in \Gamma$ such that $\phi_1(m, \sigma) = (\gamma, m_1)$ and $\phi_2(m, \sigma) = (\gamma, m_2)$, then $\phi_1 = \phi_2$.*

This property, which states that the output produced in response to any given input determines which processing function has been applied, is important for testing purposes and in practice can be easily achieved by adding, if needed, some extra output symbols.

Definition 7. *An SXM Z is called* input-complete *if $\forall \phi \in \Phi, m \in M, \exists \sigma \in \Sigma$ such that $(m, \sigma) \in dom(\phi)$.*

The input-completeness (or controllability) of an SXM assures that any sequence of of processing functions in the associated finite automaton (FA) can be

triggered by suitable input sequences, so they can be tested against the implementation. This property is rather strict; and most specifications corresponding to real systems are not by default input-complete.

Definition 8. *A* test function *of an SXM Z is a function $t : \Phi^* \longrightarrow \Sigma^*$ that satisfies the following conditions:*

- $t(\varepsilon) = \varepsilon$ (1)
- *Let* $p = \phi_1 \dots \phi_k \in \Phi^*$, $k \geq 1$
  - *Suppose* $\phi_1 \dots \phi_{k-1} \in L_{A_Z}$ *and there exists* $\sigma_1, \dots, \sigma_k \in \Sigma$, $\gamma_1, \dots, \gamma_k \in \Gamma$ *and* $m_1, \dots, m_k \in M$ *such that* $\phi_i(m_{i-1}, \sigma_i) = (\gamma_i, m_i)$, $1 \leq i \leq k$. *Then* $t(p) = \sigma_1 \dots \sigma_k$ *for some* $\sigma_1 \dots \sigma_k$ *that satisfy this condition* (2)
  - *Otherwise,* $t(p) = t(\phi_1 \dots \phi_{k-1})$. (3)

As the initial design conditions for DSXM integration testing were quite restrictive, in further works they have been relaxed, for example the *controllability* has been replaced by *input-uniformity* in [11]. This property of the X-machine suggests that, having a sequence of processing functions in the FA, one can determine an input sequence that drives this sequence of functions by simply selecting appropriate input symbols for each processing function in the sequence, one at a time, without needing to know the processing functions to be applied next.

2.4 The Test Suite

As mentioned previously, the test suites derived from X-machines are mainly inspired from FSM testing and many derivation approaches are extensions of the W-method.

After applying a certain methodology, some "abstract" test sequences in the associated FA are produced and in order to obtain a corresponding test suite in the X-machine, the input values that trigger the processing functions should be generated according to Definition 8.

It might be the case that the sequences produced are not all feasible and consequently they cannot be mapped into a set of transitions with actual input values. In this case, as Definition 8 case (3) specifies, only the longest subsequence will be mapped into actual input values.

However, in the SXM-testing literature there is no suggestion how to automatically obtain these input values. The problem of how to generate feasible paths and their corresponding input values is in general an NP-complete problem. This is why a tool that automatically generates these complete test suites is desirable, first, because one could rely on the theoretical results that assure the quality of test suite derived according to the SXM testing methodologies and, second, the human effort would be limited to writing good SXM specifications.

3 Tool Presentation

Based on the SXM testing framework, a tool[3] has been developed, in order to help users specify X-machine models, verify, validate them and automatically

[3] `http://staffwww.dcs.shef.ac.uk/people/A.Simons/broker/`

generate high-level functional test suites from these specifications. The tool aims to be used for testing cloud services, after specifying them using the DSXM formalism.

The software developed consists of three main modules:

- *The validation component*, responsible for checking the X-machine specified and announce the user if there are non-reachable states or missing transitions.
- *The verification component*, responsible mainly for checking the operations of the service protocol, in order to assure determinism.
- *The test generation component*, which will deliver high quality test suites, providing also the actual value for each parameter of the processing functions.

3.1 Validation Component

The XML specification of a service consists firstly of a functional part, which defines the constants and variables used in the service model, the signatures of the service's operations and their inputs, outputs, branching conditions and state update effects on the memory variables [14]. For a complete description of the service specification language, the BNF of the language may be consulted [14].

The second part of the XML specification is represented by a state machine, which captures the high-level control states of the service and shows its allowed transitions, labelled with the names of distinct request/response (event/action) pairs taken from the operations [14], which correspond to the processing functions of the SXM.

For example, an excerpt from a web service specification representing the system for Book reservation and borrowing from Fig. 1 would look like this:

```
<State name="Available" initial="true">
 <Transition name="borrow/ok" source="Available" target="Borrowed"/>
 <Transition name="borrow/error" source="Available" target="Available"/>
 <Transition name="reserve/ok" source="Available" target="Reserved"/>
 <Transition name="reserve/error" source="Available" target="Available"/>
</State>
```

The validation module allows users to cross-check the design of their state machine against the service's operations, and to determine whether all events are appropriately handled.

The output of this module is an XML file containing the analysed state machine specification. The root Machine node will contain a *Notice* node, which may contain further Analysis or Warning nodes. For example

```
<Notice id="1" text="Validation report for machine: BookServiceMachine">
 <Analysis id="2" text="Events are ignored in state: Available"/>
   ...
</Notice>
```

A *Warning* is issued if there are states that cannot be reached in the machine, or if known events (service's operations) are not handled by the machine. These are *faults* in the specification that should be rectified. An *Analysis* is issued if any state ignores certain events. This is not necessarily a fault, and it is provided for further information.

A *Notice* is then also attached to each State node, giving an Analysis of which events are ignored by that state. A state may legitimately choose to ignore certain events; but the analysis allows the user to check that events are handled as it was intended. The Warning is repeated for each unreachable state, as a reminder.

```
<State id="6" name="Available" initial="true">
  <Notice id="7" text="Completeness check for state: Available">
    <Analysis id="8" text="State ignores the events:">
       <Event id="9" name="return/ok"/>
       <Event id="10" name="return/error"/>
       <Event id="11" name="cancel/ok"/>
       <Event id="12" name="cancel/error"/>
    </Analysis>
  </Notice>
```

3.2 Verification Component

This module is responsible for checking a specification for *completeness* and *consistency*. As explained before, some operations of the service can have different behaviours or branches, depending on the values of the operation's inputs, or internal memory variables. For example, the withdraw operation from a bank account can have different branches, for error (when the amount requested is higher than the current balance) or for success (when the value is less or equal to balance). These different behaviours are specified in the XML model of the X-machine as "scenarios" of the same operation, corresponding to different processing functions of the X-machine. The difference between these scenarios is given by the *condition* of each one, which is an expression restricting the domain of the processing function, for example an excerpt of the account service contains the following:

```
<Scenario name="withdraw/ok">
 <Condition>
  <Proposition name="and">
    <Comparison name="moreThan">
      <Input name="amount"/>
      <Constant name="zero"/>
    </Comparison>
    <Comparison name="notMoreThan">
      <Input name="amount"/>
      <Variable name="balance"/>
    </Comparison>
  </Proposition>
```

```
 </Condition>
 ...
</Scenario>
```

The verification module checks each operation, to ensure that whatever the current state of variables in memory, and whatever values are supplied as inputs, there will always be exactly one path that is executable.

The verification process uses symbolic evaluation to determine possible partitions in the input space and symbolic subsumption to check which path is enabled. The verification process can detect whether the specification exhibits *nondeterminism* (more than one scenarios enabled, this is equivalent to several processing functions having overlapping input domains) or *blocking* (no path is enabled, or no function can process the input) for particular inputs and states.

The output of the verification is an XML file containing the analysed protocol specification. The root Protocol node will contain a *Notice* node, which may contain further Analysis or Warning nodes. An *Analysis* node is issued if the memory is correctly initialised and each operation is found to be deterministic. A *Warning* node is issued if the memory is not fully initialised, operation inputs are not all bound, or known events are not handled by the protocol specification. A *Warning* is also issued if an operation is found to be blocking, or non-deterministic under certain inputs. These warnings indicate faults in the specification that should be corrected. For example:

```
<Notice id="1" text="Verification report for protocol: BookService">
  <Analysis id="2" text="Memory is correctly initialised"/>
  <Warning id="3" text="Operation is blocking: borrow(id)"/>
  <Warning id="4" text="Operation is blocking: reserve(id)"/>
  <Analysis id="5" text="Operation is deterministic: return(id)"/>
  <Analysis id="6" text="Operation is deterministic: cancel(id)"/>
</Notice>
```

A *Notice* node is then attached to each Operation node, describing the analysed behaviour of that operation. If the operation is sensitive to different inputs, another Notice node is attached, containing an Analysis node for each input partition. Otherwise an Analysis node reports that the operation accepts universal input. Then, for each input partition, an Analysis node is created if one scenario accepts this input; and a Warning node is issued if no scenarios accept the input (blocking); or if multiple scenarios accept the input (non-determinism) - the multiple scenarios are identified in appended Analysis nodes under the warning. In blocking and non-deterministic cases, the specification is faulty and should be corrected.

3.3 Test Generation Module

This module aims to systematically explore the paths through a specification, according to the methodology presented in Section 2 and to generate complete test suites from a cloud software service specification.

High-Level Test Suite Generator. The internal mechanism of the module is described in the following. It will generate first a reachable state cover set S_r for the X-machine (which can be different from the one of the associated automaton, which in case of X-machines could be infeasible). Then, a transition cover $T = S_r \cup S_r\Phi$ is generated and, according to the W-method, a test set $Y = T\Phi[k]W$ should be constructed. In practice, if the current state can be checked using an abstract or a precise oracle, the W characterisation set can disappear from the previous formula, resulting in a reduction in test set size. This is at the cost of inserting a call to the oracle function to inspect the reached state. The tool considers then for generation only sets of the form $Y' = S_r\Phi[k]$. Test sequences are generated to a finite bounded depth k, given by the user, to avoid an explosion of cases.

The maximum path depth k can be arbitrarily chosen. When $k = 0$ the generated test suite is the state cover; and when $k = 1$, it is the transition cover of the associated automaton. Since all complete paths through the FA are not necessarily realizable in the SXM, the tool warns the user if states are not reached, or transitions not covered, for any given value of k.

```
<Warning id="6" text="Specification is not fully covered by the test
  suite">
  <Analysis id="7" text="Suggest increasing the path length"/>
</Warning>
<Warning id="8" text="These transitions were not tested:">
  <Transition id="9" name="borrow/error" source="Available" target=
     "Available"/>
  <Transition id="10" name="reserve/error" source="Available" target=
     "Available"/>
  <Transition id="11" name="return/ok" source="Borrowed" target=
     "Available"/>
  ...
```

The output will be an XML file containing the high-level tests, its root node *TestSuite* will contain a *Notice* node, whose children nodes consist of Advice nodes describing the stages in generating and filtering the resulting tests, and *Warning* nodes describing any transitions and states that were not covered in the specification, for the chosen depth k of exploration. The remaining *TestSequence* nodes are the paths to test, presented as an ordered set, for example:

```
<TestSequence id="39" state="BorrowedReserved" path="0">
  <TestStep id="40" name="create/ok" state="Available">
    <Operation id="41" name="create"/>
  </TestStep>
  <TestStep id="42" name="borrow/ok" state="Borrowed">
    <Operation id="43" name="borrow">
      <Input id="44" name="id" type="Integer">1</Input>
      <Output id="45" name="result" type="Boolean">true</Output>
    </Operation>
  </TestStep>
```

```
  <TestStep id="46" name="reserve/ok" state="BorrowedReserved"
    verify="true">
    <Operation id="47" name="reserve">
      <Input id="48" name="id" type="Integer">2</Input>
      <Output id="49" name="result" type="Boolean">true</Output>
    </Operation>
  </TestStep>
</TestSequence>
```

Each TestSequence describes a unique scenario to test, consisting of a sequence of TestSteps. Typically, the early *TestSteps* in a sequence denote set-up actions and the final TestStep is the particular step under observation, to be verified. However, if multi-objective tests were selected, some TestSequences may also have intermediate verified TestSteps. This is where shorter tests have been merged into longer tests, where the shorter sequence is a prefix of the longer sequence.

Low-Level Test Suite Grounding. The returned high-level test suite is intended for service providers to develop their own bespoke grounding to concrete tests. A grounding is a transformation from high-level abstract tests to low-level concrete, executable tests. This may be performed by programs that understand the XML format of the generated high-level tests, and the expected implementation technology of the cloud service to be tested.

It is fairly simple to build a grounding and the process is described in [14]. Each TestSequence starts with a freshly-created (or reset) instance of the service. The TestSequence then describes a particular scenario to test, shown as a sequence of TestStep interactions. Each TestStep represents a single interaction with the service, in which a given operation is called with particular concrete inputs that should trigger the given scenario. Typically, the early TestSteps in a sequence constitute the set-up actions for the desired scenario and the final TestStep is the step to be verified. Whenever a TestStep is flagged for verification, then the grounding should generate code to verify, through assertions, that:

- a request triggers the expected named scenario in the operation
- the service subsequently enters the state named by the step
- any outputs returned in the response are the expected outputs

Positive tests are indicated by the scenario's response-name, for example, *ok* for a normal response, or *error* for a planned error handler. Negative test sequences are always indicated by the response *ignore*, which is generated automatically. No outputs are simulated for negative tests, but the implementation should flag that the request was refused. Negative tests always arise from refusals by the state machine, since failing a guard for one scenario always triggers a different scenario (positively), so long as the guards are mutually exclusive and exhaustive.

In the above, we presume that software services are designed respecting the mentioned design-for-test criteria. Each response from a service must indicate, in

addition to the usual returned value, metadata about which branch of an operation (scenario) executed, and what abstract state the service entered afterwards. This information could be supplied as extra header information, or as additional query-operations upon the service, which could be interleaved with the tested operations above.

4 Example XML Service Specifications

The testing tool is offered as a web service, able to receive XML specifications of the cloud software services under test. The web standard for specifying the functional behaviour of the system under test is also provided and it has been presented in [14]. To assist users understanding how a valid service specification may be defined according to this schema, several XML service specifications are also given as examples, as valid implementations of the schema and provided on the tool web page [4]. Due to space constraints we will briefly summarize them, more details and complete specification can be found online.

- A simple *login service*, with two states and no memory updates, illustrating a state machine, guarded scenarios and default ignore transitions, inserted during test generation.
- A bank *account service*, with two states and integer balance updates in memory. This example illustrates how to initialise and update memory variables and design guards with compound conditions.
- A *contact list service* with data stored as two lists in memory. This is a more complex example, illustrating the need to complete functions with enough scenarios to trigger all memory-dependent branches; more variants are provided in order to illustrate how a specification can be improved for testability reasons.
- A simple *shopping cart* that represents the state of the cart and the level of stock as two maps in memory. This example illustrates the pure functional style of updates to data structures in memory.
- A simple *book lending service*, modelling the constraints on borrowing and reserving books. This example illustrates the discovery of incomplete operations, where the given scenarios do not cover all input and memory cases.
- A simple *Cloud data storage service*, with document versioning and a limit set on the volume of data to be stored. This example illustrates the use of an extra local variable in memory to assist with version numbering.

5 Related Work

Different approaches to cloud service testing can be consulted in two excellent surveys about testing and verification of service oriented architectures [5, 3]. Among these, only a few only a few approaches have used the state-based nature

[4] `http://staffwww.dcs.shef.ac.uk/people/A.Simons/broker/specify.html`

of services for testing purposes; and this research has not led to a mature testing tool.

The current paper is extending the work from [14] in the following aspects. It introduces the theoretical X-machine background, needed to understand the test suite generation mechanism and the properties (or design for test conditions) the specifications should satisfy. Secondly, compared to previous variant of the tool, the *validation* and the *verification* modules have been added, in order to help the user design proper specifications. Also, some improvements in the test generation algorithm, regarding the search for input values that trigger every transition on the path, have been realized.

Another tool, called JSXM [8], was developed and used for SXM based testing, using the theoretical foundations from [12]. JSXM supports the animation of SXM models, described in an XML-based language with Java in-line code, and the automatic generation of test suites from the SXM specifications. However, the user modelling the system should provide the r-state cover S_r and a separating set W_s (its construction for more complex machines can be tedious).

6 Conclusions and Further Work

This paper presents a tool for model based testing cloud services, using as formal specification the stream X-machine model. It summarizes the theoretical background of SXM testing and explains what properties the specification must satisfy in order to obtain high quality test suites.

Future work will focus on automatic groundings for certain standard service implementation technologies. Another interesting research direction is to apply metaheuristic search algorithms to generate the concrete input parameter values which can trigger the given functions from a generated sequence.

Acknowledgements. 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. Binder, R.V.: Testing Object-oriented Systems: Models, Patterns, and Tools. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1999)
2. Bogdanov, K., Holcombe, M., Ipate, F., Seed, L., Vanak, S.K.: Testing methods for X-machines: a review. Formal Asp. Comput. 18(1), 3–30 (2006)
3. Bozkurt, M., Harman, M., Hassoun, Y.: Testing and verification in service-oriented architecture: a survey. Softw. Test., Verif. Reliab. 23(4), 261–313 (2013)
4. Briand, L.C., Penta, M.D., Labiche, Y.: Assessing and improving state-based class testing: A series of experiments. IEEE Trans. Software Eng. 30(11), 770–793 (2004)
5. Canfora, G., Penta, M.D.: Service-oriented architectures testing: A survey. In: Lucia, A.D., Ferrucci, F. (eds.) ISSSE. Lecture Notes in Computer Science, vol. 5413, pp. 78–105. Springer (2008)

6. Chow, T.S.: Testing software design modeled by finite-state machines. IEEE Trans. Software Eng. 4(3), 178–187 (1978)
7. Coakley, S., Gheorghe, M., Holcombe, M., Chin, S., Worth, D., Greenough, C.: Exploitation of high performance computing in the FLAME agent-based simulation framework. In: Min, G., Hu, J., Liu, L.C., Yang, L.T., Seelam, S., Lefevre, L. (eds.) HPCC-ICESS. pp. 538–545. IEEE Computer Society (2012)
8. Dranidis, D., Bratanis, K., Ipate, F.: JSXM: a tool for automated test generation. In: Eleftherakis, G., Hinchey, M., Holcombe, M. (eds.) SEFM. Lecture Notes in Computer Science, vol. 7504, pp. 352–366. Springer (2012)
9. Eilenberg, S.: Automata, Languages, and Machines. Academic Press, Inc., Orlando, FL, USA (1974)
10. Holcombe, M., Ipate, F.: Correct systems - building a business process solution. Applied computing, Springer (1998)
11. Ipate, F.: Testing against a non-controllable stream X-machine using state counting. Theor. Comput. Sci. 353(1-3), 291–316 (2006)
12. Ipate, F., Holcombe, M.: Testing data processing-oriented systems from stream X-machine models. Theor. Comput. Sci. 403(2-3), 176–191 (2008)
13. Ipate, F., Holcombe, M.: An integration testing method that is proved to find all faults. International Journal of Computer Mathematics 63, 159–178 (1997)
14. Kiran, M., Friesen, A., Simons, A.J.H., Schwach, W.K.R.: Model-based testing in cloud brokerage scenarios. In: Lomuscio, A., Nepal, S., Patrizi, F., Benatallah, B., Brandic, I. (eds.) ICSOC Workshops. Lecture Notes in Computer Science, vol. 8377, pp. 192–208. Springer (2013)
15. Laycock, G.T.: The Theory and Practice of Specification Based Software Testing. Ph.D. thesis, University of Sheffield (1983)
16. Zhang, Q., Cheng, L., Boutaba, R.: Cloud computing: state-of-the-art and research challenges. J. Internet Services and Applications 1(1), 7–18 (2010)