JSXM: A Tool for Automated Test Generation

Dimitris Dranidis[1], Konstantinos Bratanis[2], and Florentin Ipate[3]

[1] The University of Sheffield International Faculty, CITY College,
Computer Science Department,
3 Leontos Sofou, 54626, Thessaloniki, Greece,
dranidis@city.academic.gr
[2] South-East European Research Centre (SEERC),
International Faculty, The University of Sheffield,
24 Proxenou Koromila, 54622, Thessaloniki, Greece,
kobratanis@seerc.org
[3] University of Pitesti,
Department of Computer Science and Mathematics,
Str. Targu din Vale 1, 110040 Pitesti, Romania,
florentin.ipate@ifsoft.ro

Abstract. The Stream X-machine (SXM) is an intuitive and powerful modelling formalism that extends finite state machines with a memory (data) structure and function-labelled transitions. One of the main strengths of the SXM is its associated testing strategy: this *guarantees* that, under well defined conditions, *all* functional inconsistencies between the system under test and the model are revealed. Unfortunately, despite the evident strength of SXM based testing, no tool that convincingly implements this strategy exists. This paper presents such a tool, called JSXM. The JSXM tool supports the animation of SXM models for the purpose of model validation, the automatic generation of abstract test cases from SXM specifications and the transformation of abstract test cases into concrete test cases in the implementation language of the system under test. A special characteristic of the modelling language and of the tool is that it supports the specifications of flat SXM models as well as the integration of interacting SXM models.

Keywords: model-based testing, automated test generation, functional conformance, incremental testing, stream x-machines, implementation, tool

1 Introduction

Performing systematic software testing manually is a time consuming and costly process, because it requires the manual definition, maintenance and execution of appropriate test cases. The manual definition of test cases can be increasingly complex, often resulting in test cases that may not provide full coverage of the implementation. Also, manual software testing is prone to human errors, which may be introduced either in the selection or execution of test cases. Consequently,

there is a clear need for automating software testing. The automation increases the reliability of the software testing process, because there is minimum human involvement in the generation and the execution of test cases.

One approach to the automation of software testing is model-based testing (MBT). MBT enables the automatic generation of test cases using models that specify the expected behaviour of the implementation under test (IUT). If MBT relies on a formalism for behavioural modelling and test case generation, various properties can be guaranteed for the IUT, such as functional conformance. One of the popular approaches to MBT is based on state-based descriptions, which are typically used to model the control flow of a system.

Stream X-machine (SXM) [9] is a state-based formalism capable of modelling both the data and the control of a system. SXMs are special instances of the X-machines introduced by Eilenberg [8]. SXMs extend finite state machines by incorporating memory and processing functions instead of simple labels. The powerful modelling capabilities of SXMs have been used in a number of research projects, such as the EURACE, SUMO and Epitheleome Project[1], for the simulation of cellular and social systems. Furthermore, SXMs have the significant advantage of offering a testing method [12, 9] that, under certain design-for-test conditions, ensures the conformance of an IUT to its specification.

Although there have been several improvements to the SXM testing method with the aim to relax the design-for-test conditions [11, 13], there exists no tool that demonstrates the practical benefits of the method. The small number of existing tools [16, 17, 15] are concerned with the modelling and the animation of SXM models, but not with the use of SXMs for automated software testing.

In this paper, we present JSXM, a new (and, to the best of our knowledge, the only existing) suite of tools that supports automated model-based test generation using SXMs. JSXM supports animation of SXM models, model-based test generation and test transformation. The test generation is based on the SXM testing method and, given an SXM model, it generates a set of test cases in XML format, which are independent of the programming language of the implementation. Test transformation is used for transforming the general test cases to concrete test cases in the underlying technology of the implementation. Currently, a JUnit transformer is available , which generates JUnit test cases.

The rest of the paper is organised as follows. Section 2 presents the SXM formalism and Section 3 the SXM testing method and the improvements of the method to relax the design-for-test conditions. Section 4 presents JSXM and describes how the SXM testing method has been implemented. Section 5 discusses the evaluation of JSXM in terms of effectiveness of the generated test cases, efficiency of the test generation process and the validation of JSXM in existing applications. Section 6 outlines the existing related tools for SXMs. Finally, Section 7 concludes the paper and presents some future directions.

[1] http://www.flame.ac.uk

2 Stream X-Machines

A Stream X-machine [9] is a computational state-based model capable of modelling both the data and the control of a system. SXMs extend finite state machines with two important additions: (a) the machine has some internal storage, called *memory*; and (b) the transition labels are not simple symbols, but *processing functions* that represent basic operations that the machine can perform. A processing function processes inputs from an input stream and produces outputs on an output stream while it may also change the value of the memory.

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

- Σ *is a finite set of* input *symbols and* Γ *is a finite set of* output *symbols;*
- Q *is a finite set of* states*;*
- M *is a (possibly) infinite set called* memory*;*
- Φ *is a finite set of partial functions* ϕ *(called* processing functions*) that map memory-input pairs to output-memory pairs,* $\phi : M \times \Sigma \to \Gamma \times M$*;*
- F *is the* next-state *partial function,* $F : Q \times \Phi \to Q$*;*
- $q_0 \in Q$ *and* $m_0 \in M$ *are the* initial state *and* initial memory *respectively.*

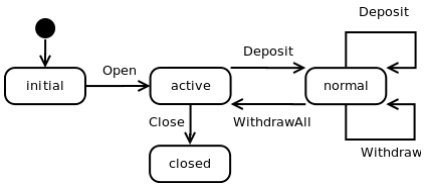
An SXM is usually graphically represented by the state diagram of its associated finite state automaton: $A_Z = (\Phi, Q, F, q_0)$.

Example 1. Fig. 1 illustrates the associated automaton of an SXM model of a system that allows performing basic operations on a bank account. For simplicity, let us assume that the system's interface comprises four operations: *open*(), *deposit*(a), *withdraw*(a), and *close*(), where a is a positive number. These comprise the input alphabet Σ of the SXM.[2] The memory of the system consists of a single number, the available balance b. Initially an account is inactive (state: *initial*) and needs to be opened (state: *active*) with its balance set to zero before any transaction can be performed. Depositing an amount results in increasing the balance (state: *normal*), while the withdrawal of an amount can only take place if the amount does not exceed the balance. An account can be closed only if its balance is zero and once closed (state: *closed*) it cannot be re-activated. The above requirements are specified in the state diagram and in the definition of its processing functions (shown on the right of the diagram in Fig. 1).

The next state function F can be extended to the function $F^* : Q \times \Phi^* \to Q$ receiving sequences of processing functions (paths in the associated automaton). The language accepted by the associated automaton of Z from state q is defined as $L_{A_Z}(q) = \{p \in \Phi^* \mid (q, p) \in \text{dom } F^*\}$ and the language accepted by the automaton is notated as $L_{A_Z} = L_{A_Z}(q_0)$.

An SXM is *deterministic* if for every state, memory, input combination there is at most one possible transition, i.e. for every $\phi_1, \phi_2 \in \Phi$ such that $(q, \phi_1) \in \text{dom } F$ and $(q, \phi_2) \in \text{dom } F$ for some $q \in Q$ either $\phi_1 = \phi_2$ or $\text{dom } \phi_1 \cap \text{dom } \phi_2 = \emptyset$. In this paper we only consider deterministic SXMs. It can be checked that the

[2] Processing function names are capitalized to easily distinguish from input symbols.



$Open(b, open()) = (openOut, 0)$
$Deposit(b, deposit(a)) =$
$\quad (depositOut, b + a)$
$Withdraw(b, withdraw(a)) =$
$\quad (withdrawOut, b - a)$ if $a < b$
$WithdrawAll(b, withdraw(a)) =$
$\quad (withdrawOut, 0)$ if $a = b$
$Close(b, close()) = (closeOut, b)$

Fig. 1. A state transition diagram for an SXM modelling a bank account and its processing functions (e.g., *Close*), which are triggered by input symbols (e.g., *close*()).

account SXM is deterministic, since $Withdraw$ and $WithdrawAll$, which both accept the input $withdraw(a)$ at state *normal*, have disjoint domains.

If $p \in \Phi^*$ is a sequence of processing functions then the function $\|p\| : M \times \Sigma^* \rightarrow \Gamma^* \times M$ associates a memory value and a stream of input symbols with the corresponding stream of output and final memory produced by following p.

An SXM Z is *completely-defined* if every sequence of inputs $s \in \Sigma^*$ is processed by at least one sequence of functions $p \in L_{A_Z}$ accepted by the associated automaton. Any SXM that is not completely defined can be transformed into one that is completely defined by adding at each state and for each not accepted input σ a self-transition labelled with the processing function σ_{error}. To make the account SXM completely defined we need to add the processing functions: $Open_{error}$,$Deposit_{error}$, $Withdraw_{error}$, $Close_{error}$, and use these as labels for loop transitions at the states that the corresponding inputs are refused. For simplicity these transitions are not shown in the diagram.

3 The SXM Testing Method

SXMs have the significant advantage of offering a testing method that under certain design-for-test conditions ensures the conformance of an IUT to a specification. The goal of the testing method is to devise a finite test set $X \subseteq \Sigma^*$ that produces identical results when applied to the specification and the IUT *only if* they both compute identical functions. The main assumption that needs to be made for the IUT is that it consists of correct elementary components, i.e. the processing functions are correctly implemented. Furthermore, it is estimated that the number of states in the IUT is $n' \geq n$, where n is the number of states of the specification. Let $k = n' - n$.

There have been several improvements to the method with the aim to relax the design-for-test conditions. The main variants are briefly presented next.

3.1 Test Generation for Input-Complete Specifications

The original SXM testing method [12, 9] relies on the following design-for-test conditions:

– **Output-distinguishability**: Processing functions should be distinguishable by their different outputs on some memory-input pair, i.e. for every

$\phi_1, \phi_2 \in \Phi$, $m \in M$ and $\sigma \in \Sigma$ such that $(m, \sigma) \in \text{dom } \phi_1$ and $(m, \sigma) \in \text{dom } \phi_2$, if $\phi_1(m, \sigma) = (\gamma, m_1)$ and $\phi_2(m, \sigma) = (\gamma, m_2)$ then $\phi_1 = \phi_2$.
- **Controllability:** Processing functions have to be input-complete w.r.t. memory, i.e. for each $m \in M$ there is $\sigma \in \Sigma$ such that $(m, \sigma) \in \text{dom } \phi$. This implies that for all memory values there should exist an input value that triggers the execution of the processing function.

The test generation is a two stage process: (1) the W method [2] is applied on the associated automaton A_Z to produce a set $Y \subseteq \Phi^*$ of sequences of processing functions, which are then (2) translated into sequences of inputs for Z.

Y is obtained by constructing a state cover S and a characterization set W of A_Z. $S \subseteq \Phi^*$ contains sequences to reach all states of A_Z, while $W \subseteq \Phi^*$ contains sequences to distinguish between any two distinct states of A_Z. Each sequence $y \in Y$ consists of three sub-sequences, i.e., $y = stw$, where $s \in S$ drives the automaton to a specific state, $t \in \Phi^*$ attempts to exercise transition-paths up to length of $k+1$ and w distinguishes the resulting state from any other state. Thus $Y = S\Phi[k+1]W = S(\bigcup_{0 \le i \le k+1} \Phi^i)W$. Note that Y may include legal as well as non-legal sequences of processing functions (positive and negative testing).

Definition 2. *A* test function $t : \Phi^* \to \Sigma^*$ *is a function that converts a sequence of processing functions to a sequence of inputs and is defined as:*

- $t(\epsilon) = \epsilon$
- $t(p\phi) = \begin{cases} t(p) & \text{if } p \notin L_{A_Z} \\ t(p)\sigma & \text{if } p \in L_{A_Z} \end{cases}$

where $p \in \Phi^*$ is a sequence of processing functions, $\phi \in \Phi$, and $\sigma \in \Sigma$ is a suitable input such that $(m, \sigma) \in dom\ \phi$ and $\|p\|(m_0, t(p)) = (g, m)$, i.e. m is the attained memory after executing the path p.

Thus, for a path that belongs to the language of the associated automaton, a sequence of inputs of the same length is produced. For a path that does not belong to the language of the associated automaton, the generated input sequence is one input longer than the longest accepted prefix of the sequence. The extra input attempts to exercise the first non-existing transition. Such an input is guaranteed to exist for every processing function if the machine is controllable.

The final test suite for checking functional equivalence is:

$$X = t(Y) = t(S\Phi[k+1]W)$$

The controllability condition is rather strict and in most practical situations specifications are not naturally input-complete. The account SXM is not controllable since $Withdraw$ and $WithdrawAll$ are only defined for $m > 0$ and $m > 1$ respectively. If a specification is not input-complete then the specification as well as the IUT needs to be augmented with extra inputs. However, the introduction of the extra inputs might lead to several problems, since after successful testing, the extra inputs of the IUT need to be removed or hidden, with the potential danger of introducing faults to the IUT. In order to alleviate this problem, the testing method has been generalized in [11] to non-controllable specifications, as presented in the following section.

3.2 Test Generation for Input-Uniform Specifications

In [11] the notion of realizable sequences is introduced. There may exist sequences of functions that are accepted by the automaton but they cannot be driven by any input sequence. These sequences are called non-realizable.

Definition 3. *A sequence $p \in \Phi^*$ is called* realizable in q and m *if $p \in L_{A_Z}(q)$ and $\exists s \in \Sigma^*$ such that $(m, s) \in dom\ \|p\|$. The set of realizable sequences of Z in q and m is notated as $LR_Z(q, m)$. Let LR_Z be defined as $LR_Z(q_0, m_0)$.*

A state is r-reachable if it can be reached by a realizable sequence $p \in LR_Z$.

Definition 4. *A set $S_r \subseteq LR_Z$ is called a* r-state cover *of Z if for every r-reachable state q of Z there exists a unique $p \in S_r$ that reaches the state q.*

The set of memory values that can be attained at a state q is notated as $MAtt(q)$ and it consists of all memory values that are the result of realizable sequences that end at state q, i.e. $MAtt(q) = \{m \in M \mid \exists p \in LR_Z$ and $\exists s \in \Sigma^*, \|p\|(m_0, s) = (g, m)\}$.

Example 2. In the account SXM, all sequences of processing functions that are accepted by the associated automaton are also realizable. $S_r = \{\epsilon, Open$, $Open\ Deposit$, $Open\ \ Close\}$. At states *initial*, *active*, and *closed* the only attainable memory is $m = 0$; at state *normal*, $m > 0$.

Definition 5. *A set $Y \subseteq \Phi^*$* r-distinguishes *between q_1 and q_2 if for every $m_1 \in MAtt(q_1)$ and every $m_2 \in MAtt(q_2)$ there exist sequences in Y that are realizable either in m_1 and q_1 or in m_2 and q_2 but not in both of them, i.e. $LR_Z(q_1, m_1) \cap Y \neq LR_Z(q_2, m_2) \cap Y$.*

The proposed testing method has replaced the controllability condition with a less strict design-for-test condition that requires the specification to be *input-uniform*. When a specification is input-uniform, the inputs that will drive a sequence of functions can be selected one at a time. If a sequence $\phi_1, \ldots, \phi_n$ is realizable and the specification is input-uniform and $\sigma_1, \ldots, \sigma_{n-1}$ is any input sequence that drives the sequence $\phi_1, \ldots, \phi_{n-1}$ then there always exist an input σ_n that will drive the next function ϕ_n. This implies that the specific choices of the previous inputs do not influence the choice of the inputs that follow.

Since the model is not required to be controllable the test generation method cannot rely on the W method. It is based on a state counting approach involving the construction of the cross-product machine of the specification and the IUT.

Example 3. The account SXM is not input-uniform as it can be illustrated with the following example: Although the sequence $Open\ Deposit\ Withdraw$ is realizable, for instance with the sequence: $open()\ deposit(2)\ withdraw(1)$, a different choice of inputs for *Open Deposit* as $open()\ deposit(1)$ does not allow to find any input for $Withdraw$, since the balance is 1 and withdrawing the least amount of 1 is not possible (the guard condition for $Withdraw$ is $a < b$). The $withdraw(1)$ input will fire the $WithdrawAll$ function instead.

3.3 Test Generation for Generic Specifications

An even more general approach was proposed in [13], in which the only required design-for-test condition is output-distinguishability. The test generation method is also based on state counting using the cross-product automaton of the specification and the implementation.

On the other hand, the condition required for distinguishing states has been strengthened: two states will have to be *separable*, i.e. distinguished by two realizable sequences with *overlapping* domains.

Definition 6. *A pair of states* (q_1, q_2) *is* separable *if there exists a finite set of sequences* Y *such that* $\forall m_1 \in MAtt(q_1), m_2 \in MAtt(q_2)$*, there exists* $p_1 \in LR(q_1, m_1) \cap Y$ *and* $p_2 \in LR(q_2, m_2) \cap Y$ *such that* $dom\ p_1 \cap dom\ p_2 \neq \emptyset$.

Definition 7. *A set* $W_s \subseteq \Phi^*$ *is called a* separating set *of* Z *if it separates (distinguishes) between every pair of separable states of* Z.

Example 4. The state *initial* in the account SXM is separable from all other states since the sequences *Open* and $Open_{error}$ have overlapping domains; at any m they accept the sequence $open()$. A separating set that separates all state pairs for the account SXM is: $W_s = \{Open, Open_{error}, Close, Close_{error}, WithdrawAll, Withdraw_{error}\}$.

An important theoretical result presented in [13] involves the case in which S_r reaches all states of Z and W_s separates all pairs of states in Z. In that special case the testing method reduces to a variant of the W-method:

$$Y = UW_s = ((S_r\Phi[k+1]) \cap L_{A_Z})W_s$$

Furthermore, the testing method requires that all sequences of $U = (S_r\Phi[k+1]) \cap L_{A_Z}$ are realizable, i.e. it is required that $U \subseteq LR_Z$. Note that the sequences of processing functions of maximum length $k+1$ that follow the r-state cover are limited to those that are accepted by the associated automaton.

In the SXM account example the set S_r covers all states, the set W_s distinguishes between all pairs of states and U is realizable, therefore the testing method can be applied for functional equivalence.

4 The JSXM Tool

JSXM [3] is a tool, developed in Java, that allows the specification of SXM models, their animation and most importantly the automated test generation.

Animation of the model means the execution of the model by providing an input stream and observing the resulting output stream. The interactive or batch animation, which are supported by the tool, allow the model designer to validate the specification, i.e. ensure that the correct functionality is modelled.

Once a model is validated, it can be used for the automated generation of test cases. The test cases that are generated by JSXM are in XML format

and they are independent of the technology or programming language of the implementation. These general test cases can then be transformed by the JSXM tool to test cases in the programming languages of the IUT.

In the following sections we briefly describe the JSXM modelling language and the associated tool suite.[3]

4.1 The JSXM Modelling Language

The JSXM modelling language is an XML-based language with Java in-line code. This allows software engineers who are familiar with these widespread technologies to model systems in a formalism that allows the automated generation of test cases and removes the barrier of having to learn a new notation.

The states and the transitions are described in XML. An extract of the JSXM code[4] for representing the state transition diagram of Fig. 1 is provided below:

```
<states>
  <state name="initial" /><state name="active" />
  <state name="closed" /> <state name="normal" />
</states>
<initialState state="initial" />
<transitions>
<transition from="initial" function="Open" to="active" />
  <transition from="active" function="Close" to="closed" />
  <transition from="active" function="Deposit" to="normal" />
  ...
</transitions>
```

The input and the output symbols are also described in XML code. Input and output symbols can be basic symbols (such as *openRequest*) or compound symbols carrying arguments, as for example the *deposit*(*amount*) input that carries an integer argument. The types of the arguments are specified as XSD types. The modeller can extend the types with any user-defined complex XSD type. The outputs are structured in a similar way to inputs:

```
<inputs>
  <input name="open" />
  <input name="close" />
  <input name="deposit">
    <arg name="amount" type="xs:int" />
  </input>
  <input name="withdraw">
    <arg name="amount" type="xs:int"/>
  </input>
</inputs>
```

```
<outputs>
  <output name="openOut" />
  <output name="closeOut" />
  <output name="depositOut">
    <result name="amount" type="xs:int" />
  </output>
  <output name="withdrawOut">
    <result name="amount" type="xs:int" />
  </output>
</outputs>
```

The memory and the body of the processing functions are written in in-line Java code. This allows the definition of any complex Java data structure as the memory of the system.

```
<memory>
  <declaration> int balance </declaration> <initial> balance = 0 </initial>
</memory>
```

[3] The tool can be downloaded from http://www.jsxm.org

[4] The complete JSXM specification for the SXM account can be found at http://www.jsxm.org/SEFM2012

Processing functions are specified by defining their inputs, outputs, preconditions (specifying the domain of the function) and effects on the memory. For brevity only one processing function is shown. Note the *object.get_par*() approach for retrieving the parameter *par* of compound inputs and outputs.

```
<function name="Withdraw" input="withdraw" output="withdrawOut">
  <precondition> balance > withdraw.get_amount() </precondition>
  <effect>
    balance = balance − withdraw.get_amount();
    withdrawOut.set_amount(withdraw.get_amount());
  </effect>
</function>
```

4.2 Test Generation Process

The JSXM tool implements the extended W-method for generic specifications (Section 3.3) for the generation of the test set. For the test generation process the modeller needs to provide:

- a JSXM specification of the SXM model Z.
- an r-state cover S_r and a separating set W_s
- a set of input generators (explained in detail in Section 4.3).
- the estimated difference k of states between the IUT and the specification.

The test generation process consists of the following steps:

1. It is being checked that all states of Z are r-reachable by S_r.
2. The set $S_r\Phi[k+1]$ is generated.
3. The generated set is restricted to these sequences that are accepted by the associated automaton: $U = (S_r\Phi[k+1]) \cap L_{A_Z}$
4. It is being checked that all generated sequences are realizable, i.e. $U \subseteq LR_Z$.
5. The sequences of the characterization set W are attached to the generated sequences yielding: $UW_s = ((S_r\Phi[k+1]) \cap L_{A_Z})W_s$
6. The resulting set of sequences is optimized: any sequence that is a proper prefix of another sequence in the set is removed from the set, since any faulty behaviour produced by this proper prefix will be identified during the animation of the prefix as part of the longer sequence.
7. All sequences of processing functions are transformed to sequences of inputs generated by the input generators. During this step it is expected that some of the sequences will not be realizable (at their W_s postfix) as formally described in the definition of the test function.
8. Finally, all the input sequences are fed to the animator that acts as an oracle and provides the expected output sequences.

As described in Section 3.3, if W_s separates all pairs of states and all sequences in step 4 are realizable, then the resulting suite is sufficient to guarantee the functional conformance of the implementation to the specification.

4.3 Input Generators

The test function transforms sequences of processing functions to sequences of inputs. The modeller has to define for each processing function an input generator function.

An input generator function is a (partial) function $in_\phi : M \to \Sigma$ for $\phi \in \Phi$. The input generator guarantees to find an input, if the memory is in the domain of the input generator. Otherwise a suitable input does not exist to make the function fire. We could define the following input generators for the account:

$in_{Deposit}(balance) = 100$
$in_{Withdraw}(balance) = 1$ if $balance > 1$
$in_{WithdrawAll}(balance) = balance$ if $balance > 0$

The transformation of a sequence $p \in \Phi^*$ of processing functions to a sequence of inputs is recursively defined as:

- $t(\epsilon) = \epsilon$
- $t(p\phi) = \begin{cases} t(p)in_\phi(m) & \text{if } p \in LR_Z \text{ and } m \in \text{dom } in_\phi \\ t(p) & \text{otherwise} \end{cases}$

where m is the memory attained after the sequence $t(p)$, i.e. $\|p\|(m_0, t(p)) = (g, m)$. From the definition it is clear that it is being assumed that the specification is input-uniform. In what follows we discuss what are the practical limitations of this assumption and how they may be overcome.

If the specification is controllable then the processing functions are input-complete w.r.t. memory, which implies that for every possible memory value there exists an input that triggers the function and therefore in_ϕ can also be totally defined. If the specification is input-uniform and the input generator is defined for all attainable memory values, then the algorithm is able to find suitable input sequences for all realizable sequences of processing functions.

If, however, the specification is not input-uniform, the generation of inputs for realizable sequences is not generally guaranteed as it is illustrated in the following example. The input generators defined above for the account SXM would fail to generate appropriate inputs for the following long but still realizable sequence: $Open\ Deposit\ Withdraw^{100}$, in which 100 $Withdraw$s follow a $Deposit$. The sequence could still be made realizable by increasing the amount parameter of the $deposit(100)$ input that was generated for $Deposit$. So the input generators successfully generate input values up to a certain length of the function sequence.

The length of the function sequences depends on k. Even in the cases of non input-uniform specifications, the modeller can carefully design the input generators in such a way so that the specification "behaves" uniformly for all bounded-sequences that will be generated for the selected k. In most practical situations k takes small values (usually less than 5). Larger k values are not expected to reveal new errors (unless the implementation is grossly erroneous) and are computationally expensive for producing the test sets, since the number of test cases and the total length of the test set depend exponentially on k.

Nevertheless, we are investigating solutions that will generalise the input generators so that all realizable sequences can be generated. We are currently examining the possibility of applying constraint satisfaction techniques to solve this problem. If we need to generate an input sequence for: *Open Deposit Withdraw Withdraw Withdraw*, the following symbolic input sequence could be generated: $deposit(x)$ $withdraw(w_1)$ $withdraw(w_2)$ $withdraw(w_3)$, with the related arithmetic constraints that need to be solved: $x > 0, w_1 < x, w_1 + w_2 < x, w_1 + w_2 + w_3 < x$. A constraint solver could return the following values: $x = 4, w_1 = w_2 = w_3 = 1$, which would make the sequence realizable.

4.4 Generated Test Cases and Transformation

For the input-output test cases to be produced, all the input sequences are fed to the JSXM animator, which acts as an oracle, and the resulting output sequences are recorded. The resulting test cases (pairs of input and output sequences) are stored in a XML file[5] in a programming language independent format.

In order to execute the tests on the IUT, the language independent generated test cases need to be transformed to the programming language of the IUT. This is the task of a test transformer. The JSXM tool allows modellers to extend the tool by defining their own test transformer for any programming language. Currently the JSXM test suite offers a JUnit test transformer, which transforms the abstract test cases into JUnit test cases.[6]

4.5 IUT Wrappers

Since the SXM model and the IUT are at different levels of abstraction, a wrapper (or adapter) is needed that will translate IUT values back to model values. In the case of Java testing, a wrapper class consists of wrapper methods for each method of the class under test. All control flow paths within a method should produce some observable result: either a returned value or an exception. These results are then translated by the corresponding wrapper method to values that will be compared by the test engine with the model outputs.

For example, a Java method *public void open*() may perform some processing (change the internal state) but it is not expected to return any result. For the purpose of testing, however, the implementation needs to satisfy the observability design-for-test condition. A wrapper wraps the method call and, depending on the outcome, returns the expected output that will be matched with the output of the model (e.g., *openOut*). Similarly, if the *open* method throws an exception (because it could not be executed) the exception has to be caught by the wrapper and a model output should be returned (e.g., *openError*).

[5] The generated XML file can be found at http://www.jsxm.org/SEFM2012.
[6] The generated JUnit test cases can be found at http://www.jsxm.org/SEFM2012.

4.6 Interacting JSXM Models and Incremental Testing

By using interacting SXM models one can perform automated incremental testing of classes that are based on other classes.

This is achieved via a special type of parameters in SXM specifications, which is the *sxm* type. Values of this type are *instances* of SXM models. The interaction of SXM instances resembles the object-oriented method invocation: an SXM, within the effect of a processing function, may send a "message", in the form of an input symbol, to another SXM, which produces an output, which is returned to the sender SXM.

In the following example we present the memory, a processing function and its corresponding input from a borrower SXM. The borrower may borrow a book if a book is available. A separate book SXM model exists, which specifies the behaviour of a book (borrowing, returning, checking availability).

```
<memory>
  <declaration> BookSXM book; </declaration><initial> book = null; </initial>
</memory>
<inputs><input name="borrowBook" ><arg name="book" type="sxm"/></input></inputs>
<functions>
  <function name="BorrowBook" input="borrowBook" output="borrowBookOut">
    <precondition> ((BookSXM) (borrowBook.get_book())).isAvailable().result; </precondition>
    <effect>
      book = (BookSXM) borrowBook.get_book();
      book.setBorrowed();
    </effect>
  </function>
</functions>
```

Through the *BorrowBook* processing function, the borrower SXM instance sends the input *isAvailable*() to a book SXM instance. The book SXM instance is being animated with this input and an output is produced, which is returned to the borrower SXM instance.

Test sets are generated separately for each SXM and its corresponding class and programs can be incrementally tested. This incremental approach removes the need to create a separate flat model which specifies the whole system.

5 Evaluation

5.1 Validation of the JSXM Tool in Existing Applications

The JSXM tool has been used and practically validated in several application scenarios described briefly below. All the applications are from the area of service-oriented computing and Web services. The XML-based specifications of JSXM and the capability to extend the input types by user-defined XSD types facilitate easier integration with Web technologies and related XML-based Web service standards.

In [5] SXMs are utilised for modelling the behavioural specification of Web services and the SXM testing method is used for generating test cases, which verify that the service implementations conform to their specification. This approach is further elaborated in [18] where JSXM is used as the main tool in a framework for the validation and verification of Web services.

The animation capability of the JSXM tool (execution of SXM models) enables the run-time verification approach that is presented in [6]. The animator serves as an oracle that provides the expected outputs for the purpose of run-time monitoring of Web services. JSXM provides an API that allows calling the animator's methods programmatically. This API is utilised in the implementation of the run-time verification architecture described in [1].

In [7] a novel approach is proposed for just-in-time testing of conversational Web services. The aim of just-in-time testing is to detect potential problems (functional inconsistencies) in service implementations and to pro-actively trigger adaptations of the service-based application. The JSXM and its test generation capability is utilised in order to automatically generate test cases on the fly without any human intervention and to test the service before its invocation.

It is important to notice that in all these applications the SXM testing method, and its realization through the JSXM tool, managed to successfully identify errors in the implementations, even with small values of $k \leq 2$. Furthermore, although the SXM testing method is based on the assumption that the basic processing functions are correctly implemented in the IUT, the generated test set managed to reveal not only control flow errors but also errors in the implementation of the functions. This implies that practically SXM testing is also able to reveal functional mismatches in the processing functions, although this capability is not theoretically guaranteed. This capability is demonstrated in the next section on the example of the account SXM.

5.2 Effectiveness and Efficiency of the Test Generation

To demonstrate the effectiveness of the generated test set we have performed mutation testing on the Java class Account, which is the implementation of the account SXM. Mutation testing was performed with Jumble [14], which is a class level mutation testing tool that works in conjunction with JUnit. The test set generated from JSXM for $k = 0$ managed to kill all 19 mutants.[7] The mutants changed both control logic of the program (negated conditionals, boundary values in conditions) as well as assignments within program paths (arithmetic operators, assigned values).

Concerning the efficiency of the test generation process, Table 1 shows the number of test cases generated and the time required for the test generation for different values of k. The test generation was performed on a Quad Core i7, 2.5 Ghz. The absolute times in seconds are provided only as an indication.

Table 1. Number of generated test cases and generation time for different values of k.

k	0	1	2	3
number of test cases	143	841	5293	35151
time to generate (sec)	0.06	0.26	1.67	46.06

[7] A description of the mutants can be found at http://www.jsxm.org/SEFM2012.

6 Related Tools

Currently, a small number of tools is available for SXMs. Most tools are concerned with the modelling and the animation of SXM models.

The Flexible Large-scale Agent Modelling Environment (FLAME) [10] is a framework that uses SXMs for modelling agent-based systems and for generating simulations in C. The FLAME framework provides an XML-based specification language, referred as XMML, for creating specifications that serve as models of the behaviour of agents. In-line C code is used for defining the behaviour of the processing functions, similarly as JSXM uses in-line Java code. FLAME includes Xparser, which is a parser for parsing XMML, in order to automatically generate simulation programs in C that can run models efficiently on HPCs [16].

X-system [15] is a tool that facilitates the modelling and the animation of SXMs. X-system has been implemented in Prolog and uses the X-Machine Definition Language (XMDL) for writing the specifications. XMDL was later extended by XMDL-O [4], which supported an object-based notation. Both languages are supported by the X-system, which is used mainly for animation of models.

Ma et al. [17] describe a tool for SXM models and the automatic generation of tests. It is not evident if this tool is capable of generating concrete test cases (sequences of inputs), since the authors demonstrate only the generation of processing function sequences (first step of the test generation process).

To the best of our knowledge, JSXM is the only tool available for automatically generating *concrete* (executable) test cases based on the SXM testing method. Neither FLAME nor X-system are known to be able to generate concrete test cases.

7 Conclusions

In this paper we have presented the JSXM tool for automated testing. The tool builds on the SXM formalism and implements an extension of the W method for generating test sets that are able to guarantee the functional conformance of an IUT to its specification. Currently the JSXM test generation is guaranteed to work in the case of input-uniform specifications. In Section 4.3, however, we have discussed and demonstrated through an example, that the test generation can practically work even in the case of some non input-uniform specifications.

Additionally, the JSXM language allows the definition of interacting SXM model instances. This feature enables the incremental testing of object oriented programs, where each class is separately modelled as an SXM.

The tool's applicability has been demonstrated in several application scenarios from the area of Web service testing, monitoring and run-time verification.

In the future we intend to change the input generators so that the test generation works for generic specifications. We also intend to implement the test generation based on state-counting, so that the tool covers also cases of specifications in which not all states are separable.

Acknowledgement. The work of F. Ipate was supported by a grant from the Romanian National Authority for Scientific Research, CNCS–UEFISCDI, project number PN-II-ID-PCE-2011-3-0688.

References

1. Bratanis, K., Dranidis, D., Simons, A.J.H.: An extensible architecture for run-time monitoring of conversational web services. In: 3rd International Workshop on Monitoring, Adaptation and Beyond/ECOWS (2010)
2. Chow, T.S.: Testing software design modelled by finite state machines. IEEE Transactions on Software Engineering 4, 178–187 (1978)
3. Dranidis, D.: JSXM: A suite of tools for model-based automated test generation: User manual. Tech. rep., Technical Report WP-CS01-09, CITY College (2009)
4. Dranidis, D., Eleftherakis, G., Kefalas, P.: Object-based language for generalized state machines. Annals of Mathematics, Computing and Teleinformatics (AMCT) 1(3), 8–17 (2005)
5. Dranidis, D., Kourtesis, D., Ramollari, E.: Formal verification of web service behavioural conformance through testing. Annals of Mathematics, Computing & Teleinformatics 1(5), 36–43 (2007)
6. Dranidis, D., Ramollari, E., Kourtesis, D.: Run-time verification of behavioural conformance for conversational web services. In: 7th IEEE European Conference on Web Services (2009)
7. Dranidis, D., Metzger, A., Kourtesis, D.: Enabling proactive adaptation through just-in-time testing of conversational services. In: Towards a Service-Based Internet, SERVICEWAVE (2010)
8. Eilenberg, S.: Automata, languages and machines. Acad. Press, New York (1974)
9. Holcombe, M., Ipate, F.: Correct Systems: Building Business Process Solutions. Springer Verlag, Berlin (1998)
10. Holcombe, M., Coakley, S., Smallwood, R.: A general framework for agent-based modelling of complex systems. In: European Conf. on Complex Systems (2006)
11. Ipate, F.: Testing against a non-controllable stream x-machine using state counting. Theoretical computer science 353(1), 291–316 (2006)
12. Ipate, F., Holcombe, M.: An integration testing method that is proven to find all faults. International Journal of Computer Mathematics 63, 159–178 (1997)
13. Ipate, F., Holcombe, M.: Testing data processing-oriented systems from stream x-machine models. Theoretical Computer Science 403(2), 176–191 (2008)
14. Irvine, S., Pavlinic, T., Trigg, L., Cleary, J., Inglis, S., Utting, M.: Jumble java byte code to measure the effectiveness of unit tests. In: Testing: Academic and Industrial Conference Practice and Research Techniques (2007)
15. Kapeti, P., Kefalas, P.: A design language and tool for x-machines specification. In: Advances in Informatics (2000)
16. Kiran, M., Richmond, P., Holcombe, M., Chin, L.S., Worth, D., Greenough, C.: Flame: simulating large populations of agents on parallel hardware architectures. In: 9th International Conf. on Autonomous Agents and Multiagent Systems (2010)
17. Ma, C., Wu, J., Zhang, T.: Sxmtool: A tool for stream x-machine testing. In: World Congress on Software Engineering (2010)
18. Ramollari, E.: Automated verification and testing of third-party Web services. Ph.D. thesis, University of Sheffield (2012)