International Journal of Hybrid Information Technology
Vol.9, No.4 (2016), pp. 255-268
http://dx.doi.org/10.14257/ijhit.2016.9.4.22

Extending Stream X-machines with Probability Intervals to Test Web Software

Qian Zhongsheng

School of Information Technology, Jiangxi University of Finance and Economics, Nanchang, 330013, China
Email: changesme@163.com

Abstract

The essential purpose of testing is to detect and remove faults in the implementation of a system. This study presents a pragmatic usage model based on Probabilistic Stream X-machines (PSXMs for short). The PSXM, which is an extended SXM with the transitions enhanced with probability intervals, is capable of describing both the data and the control aspects of a given system. The test case generation process is given based on the PSXM usage model. It takes the self-developed SWLS (Simple Web Login System), a miniature web software as a running example to demonstrate the testing method. The testing process ensures that the failures occurring most frequently in operational uses will be found early. This testing approach proposed lends itself to exploring new model-based testing techniques.

Keywords*: web software, probabilistic stream X-machine, probability interval, test case, formal method*

1. Introduction

Web softwares [1-3] are sophisticated, interactive systems with complicated GUIs and numerous backend software components that are integrated in fire-new and unusual ways. Faulty web softwares can have far-ranging consequences on businesses, economies, health and so on. Theoretically, all the entities of web softwares must be tested thoroughly to ensure that the system is reliable and meets its original design specifications.

Web testing aims at finding errors in the tested object and giving confidence in its correct behavior by executing the tested object with selected input values. There is the famous 80-20 law (i.e., Pareto rule) [4] stating that typical software spends 80 percent of the time executing 20 percent of the code. That means some portions of software are executed with a higher frequency than others.

Testing is a means to verify whether a component conforms to certain functional or non-functional requirements. Non-functional requirements may reflect constraints on performance, security, or other quality of service properties. Functional requirements relate to the behavior that a component is expected to exhibit when consumed. Different types of testing techniques can be applied depending on the requirements and the type of conformance to be asserted, but in most of the cases, testing can only be performed in a black-box manner, since a component's source code is usually unavailable. Verifying the conformance of a component's behavior against the requirements is especially challenging in the case an interaction protocol is assumed. A number of research works have attempted to address this type of problem with the help of formal methods. A common characteristic in all of these works is the construction of a model explicating the behavior of a component, and the generation of test cases to assert that a component implementation indeed satisfies the specification requirements.

Formal methods have been greatly studied in the past few decades. They allow us not only to represent with precision the systems that we are going to study but also to reason about them with mathematical precision and rigor. It is widely recognized that formal methods and testing are complementary techniques since they help to check the correctness of systems and provide a framework for testing [5].

There is a variety of opinions concerning the efficiency of different testing technologies. General approaches to testing systems contain the following:

- *coverage-based testing*. It tests all the possible paths through the system.
- *specification-based testing*. It tests the various components/modules of the system for proper functionality. It ensures that the system performs its intended function as expected.
- *partition-based testing*. It divides the input domain of systems according to equivalent classes, and tests each class.
- *statistical testing*. It uses a formal experimental paradigm for random testing according to a usage model of systems. It is based on usage distributions ensuring that on average the most frequently used operations receive the most testing.

The abovementioned statistical testing is a useful complement to traditional testing. In statistical testing, sequences of action (input) are stochastically generated based on a probability distribution that represents a profile of anticipated use of the system. Statistical usage testing aims to identify these portions and adjusts test suites subjecting more frequently executed parts to more thorough testing. That’s to say, this testing is based on the idea that different parts of the software don’t need to be tested with the same thoroughness. It is often important, however, to ensure the exercise of specific operations of the software irrespective of their usage probabilities. Two examples are operations of high criticality due to potential impacts of a failure and those implemented by new software. This work does not concern this situation.

2. Specifying Web Software using Probabilistic SXM

It is widely accepted that models constitute an important mechanism in the process of web software development. Models help us understand web softwares by omitting some details. The choice of what to model has a significant effect on understanding a problem and suggesting the solution.

2.1. What is Probabilistic SXM

SXM, proposed by Laycock [6], is a special instance of the *X-machine* originally introduced in 1974 by S. Eilenberg [7]. *SXM* employs a diagrammatic method of modeling control flow by extending the expressive power of *FSM*. Compared to an *FSM*, instead of using abstract symbols, the labels of the transitions in a *SXM* are *relations* (often *partial functions*) that operate on a basic data set *X*. The set of these relations, Φ, is called the *type* of the *SXM* and represents the elementary operations that the *SXM* is capable of performing. In *SXM*s, all data are triples consisting of a stream of input symbols, a stream of output symbols and an internal memory value. *SXM*s are capable of modeling non-trivial data structures by employing a memory attached to the state machine. The benefit of adding the memory construct is that the state explosion is avoided and the number of states is reduced to those states that are considered critical for the correct modeling of the system’s abstract control structure. A *divide-and-conquer* approach to designing allows the model to hide some of the complexity in the transition functions, which are later exposed as simpler *SXM*s at the next level.

So, the basic idea is that the *SXM* has some internal memory, *M*, and the stream of inputs determines, depending on the current state of control and the current state of the memory, the next control state, the next memory state and the output value. *SXM*s [6, 8] are a computational model capable of representing both the data and the control of a system. The computation of the *SXM* starts in a given initial state (the control state) and a given state of the system's underlying data set *X* (the data state).

Definition 2.1 A *SXM* is a nonuple $Z = (\sum, \Gamma, Q, M, \Phi, F, I, T, m_0)$, Where:

- $\sum$ and Γ are finite sets called the *input alphabet* and *output alphabet* respectively.
- Q is the finite set of *states*.
- M is a (possibly) infinite set called *memory*.
- Φ is the *type*, a finite set of distinct non-empty *processing relations* of the form $\varphi : M \times \sum \leftrightarrow \Gamma \times M$; usually Φ is a set of (partial) functions.
- F is the (partial) *next state function* (or *state transition function*), $F: Q \times \Phi \rightarrow 2^Q$; as for finite automata, F is usually described by a *State Transition Diagram (STD)*.
- I and T are the sets of initial and terminal states respectively, $I \subseteq Q$, $T \subseteq Q$.
- m_0 is the initial memory value, $m_0 \in M$.

In *SXM*s, each processing relation will read an input symbol, then discard it and produce one or more output symbols while (possibly) changing the value of the memory.

Definition 2.2 Given a *SXM* $Z = (\sum, \Gamma, Q, M, \Phi, F, I, T, m_0)$, the finite automaton $FA_Z = (Q, \Phi, F, I, T)$ over the alphabet Φ is called the *associated finite automaton* of *Z*. Here the alphabet Φ is regarded as a set of abstract input symbols.

The modeling tool we employed is *SXM* enhanced with probability intervals defined as follows.

Definition 2.3 A *PSXM* (*Probabilistic Stream X-Machine*), which is derived from *SXM*, is a decuple $Z = (\sum, \Gamma, Q, M, \Phi, F, \Theta, I, T, m_0)$ where Θ is a set of *probability intervals*. The transition function F is redefined as $Q \times \Phi \times \Theta \rightarrow 2^Q$. For $\forall q_1, q_2 \in Q$, $\exists \varphi \in \Phi$, $\rho \in \Theta$ such that $F(q_1, \varphi, \rho) = q_2$, which indicates if the current state is q_1 and the processing relation is φ, then there will be a transition from the current state to the successive one called q_2 with the probability interval ρ.

Each probability interval ρ ($\in \Theta$) is represented as one of (α, β), $(\alpha, \beta]$, $[\alpha, \beta)$ or $[\alpha, \beta]$ where $0 \leq \alpha \leq \beta \leq 1$. If $\rho = [\alpha, \beta]$ where $\alpha = \beta$, then we simply write $\rho = \alpha$ (or $\rho = \beta$). Especially, ρ may equal to 0 or 1.

A *PSXM* gives rise to a relation between the input sequences applied to the machine and the output sequences that it produces. This relation is given by the execution of a sequence of processing functions, from the initial state of the machine that allows obtaining an output sequence in response to an input sequence. In our formalism, we will require, for dealing with the specified probability intervals, to extend this relation to take into account the probability information when the machine receives an interaction from the environment.

Definition 2.4 Let $Z = (\sum, \Gamma, Q, M, \Phi, F, \Theta, I, T, m_0)$ be a *PSXM*. A tuple $CP_Z = (q_0, \varphi, \rho, q)$ where $q_0 \in I$, is a *computation* of *Z* if there exists k (≥ 0) states $q_1, \ldots, q_k \in Q$, such that for $\forall j$ $(1 \leq j \leq k)$ we have $F(q_{j-1}, \varphi_j, \rho_j) = q_j$ and $F(q_k, \varphi, \rho) = q$. A tuple (q_0, φ, ρ, q) is a *complete computation* if $q \in T$.

Definition 2.5 Let $Z = (\sum, \Gamma, Q, M, \Phi, F, \Theta, I, T, m_0)$ be a *PSXM*. A tuple $CF_Z = (m, q, s, g)$, where $m \in M$, $q \in Q$, $s \in \sum^*$, $g \in \Gamma^*$, is a *configuration* of *Z*. The *PSXM* starts its computation from an *initial* configuration (m_0, q_0, s, τ), where $q_0 \in I$, the initial input stream is set to s and τ indicates that the output stream is empty. A *change of configuration* $(m, q, \sigma s', g) \Rightarrow (m', q', s', g\gamma)$ is possible if there exists $\varphi \in \Phi$ and $\rho \in \Theta$ with $q' \in F(q, \varphi, \rho)$ and $\varphi(m, \sigma) = (\gamma, m')$. A configuration (m, q, τ, g) is *final* if the input stream is empty and $q \in T$ representing successful termination, where all the input streams are consumed.

A complete computation is just the process from an initial configuration (m_0, q_0, s, τ) to a *final* configuration (m, q, τ, g) through diverse changes of configurations. Intuitively, the computation is a sequence of processing functions corresponding to the transitions of a chain of configurations. The first of these configurations begins with the initial state of the machine.

Apart from being formal and proven to possess the computational power of Turing machines [8], *SXM*s offer a highly effective test method for verifying the conformance of a system's implementation against the specification. *SXM* models can be represented in *XMDL* (X-Machine Definition/Description Language), a special-purpose markup language introduced by E. Kapeti and P. Kefalas [9], which is intended to be an ASCII-based interchange language. The use of *XMDL* makes formal specification more practical, since models specified in this language can be processed by various tools, such as an animator, a test set generator, a model checker etc. *XMDL* has served as a common language for the development of numerous tools supporting *SXM*s [10].

2.2. How to Model Web Software

Models are useful for clarifying requirements and specifying implementation behavior. When a model is formal, it can also be used to generate design or implementation components, and can be verified for properties such as broken links or length of web navigation path. The model is extremely important for formalizing web software.

Generally, we model the system as a computable machine. In the machine diagram, each transition would then be implemented as a function (processing relation) of type *InternalState* $\rightarrow$ *InternalState*. We call such a structured state model an *InternalState*-machine (i.e., a machine that manipulates values of type *InternalState*) and more often, if the internal state can be represented by an object of type *X*, we call the model an *X*-machine. Everyday *X*-machines contain such examples as simple calculators (*Number*-machines), word processors (*Document*-machines) and air conditioners (*Temp*-machines).

*SXM*s allow both control and data processing of a system to be depicted separately, since the control processing is depicted by inputs, outputs, states and transitions, similarly to simple *FSM*, whereas the data processing is depicted by a memory and a set of functions. The memory acts as a global set of values, which can be retrieved and updated at any time and represents the main data that can be processed by the machine. The set of functions is the primary data processing unit of the *SXM*. In order to handle the data, the functions either can perform alterations to the memory on their own or can call external data processing functions to do so. The latter are essential for a *SXM*, since they establish a communication between the *SXM*'s control processing and the data processing unit.

The *PSXM* usage model can provide a powerful way to generate test cases for web software. We employ the self-developed SWLS (Simple Web Login System), a miniature web software to demonstrate our approach whose state transition diagram of its *PSXM* is depicted in Figure 1. The numerical range below each processing relation represents the probability interval from the source state to the target state.

Starting from the first page (indicated by a dashed arrow, reasonably, a *blank* page can be used to request for the first page of a web software), i.e., a *login page* (q_1), the user enters the *userid* and *password*, and then presses the *submit* button. Upon this pressing, the *userid* and *password* are sent to the web server for authentication. A *logged page (q_2)* will be loaded if both *userid* and *password* are correct. On the contrary, an *error page (q_3)* containing its error message is displayed if at least one of the submitted values for *userid* and *password* is wrong. From the *logged page*, it is possible to go to *info page (q_4)* for secure information viewing by just clicking on the *browse* link. The user can click on the intra-page link *continue* to view the different parts of the same page q_4 if it is too long. A *logout page (q_5)* will be displayed when the user presses the *exit* or *logout* button. Then, the user may come back to the *home page* for login again. Note that each time the *login page* is displayed, then both the *userid* and *password* fields should be initialized to be empty.

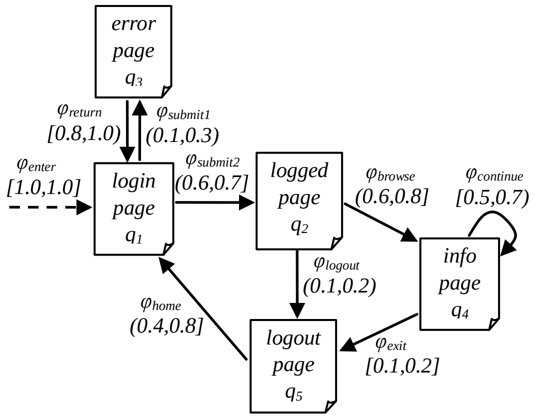
error page q_3
φ_{return} [0.8,1.0)
$\varphi_{submit1}$ (0.1,0.3)
φ_{enter} [1.0,1.0]
login page q_1
$\varphi_{submit2}$ (0.6,0.7]
logged page q_2
φ_{browse} (0.6,0.8]
$\varphi_{continue}$ [0.5,0.7)
info page q_4
φ_{logout} (0.1,0.2)
φ_{home} (0.4,0.8]
logout page q_5
φ_{exit} [0.1,0.2]

Figure 1. The State Transition Diagram of the PSXM for SWLS

We can see from Figure 1, a *PSXM* usage model comprises a unique start state (i.e., the invoked state that represents invocation of the system) that is one of the initial states, a set of final states (i.e., the terminate state that represents termination of the system), a set of intermediate usage states, and transitions between states. User actions are represented as state transitions in the *PSXM* usage model. Conventionally, a transition is linked to an input from environment of the system under test. The probability that a user performs a given action (input) in a particular state is represented by the associated transition probability in the *PSXM* usage model. Given the present state, the *PSXM* requires that the next state is independent of all past states.

In fact, the usage model of a system can be obtained from its explicit *SXM* model. The only missing information to make it a *PSXM* is an estimate of the transition probabilities to be associated with the transitions. Values for such probabilities can be computed by using historical information. It represents the conditioned probability of moving from the current system state toward another state in the next step.

As discussed above, the *PSXM* usage model originates from *SXM* enhanced with probability intervals, so the resulting model typically describes the state of usage rather than the state of the web software. The maximal probabilities of all transitions leading away from one state need to add up to 1. However, the maximal probabilities from some of the states are not added up to 1 in Figure 1, because there may be other situations, in which the user quits the web software from the given state without following the normal navigations.

The user can, for example, traverse to *info page (q_4)* via φ_{browse} with the maximal probability 0.8, to *logout page (q_5)* via φ_{logout} with the maximal probability approximating 0.2 from *logged page (q_2)*. The sum of the values of the maximal probabilities from *logged page*

(q_2) is less than 1.0 (0.8+0.2), but not equal to 1. This means that other events may also happen. For example, the user quits the web software by just closing *logged page (q_2)* abnormally. Web browsers can also provide additional navigation functions that are out of control of the web pages, such as scrolling, back, forward buttons and a history list, or even URL rewriting.

The state transitions of a *PSXM* can be represented as a two-dimensional matrix called *probability transition matrix* with the direct transition probability intervals as entries. The "direct" here means that any transition from one state to another need not pass any medial state. That's to say, the probability transition matrix has a probability interval as its *(i, j)* entry when $\varphi : q_i \to q_j$ holds and a $\perp$ in this position if $\varphi : q_i \to q_j$ does not hold for any φ. The symbol $\perp$ means that q_j can not be directly reached from q_i, i.e., q_j is not the successor state of q_i.

Therefore, the probability transition matrix of *PSXM* usage model shown in Figure 1 can be constructed as

$$\begin{bmatrix} \perp & (0.6, 0.7] & (0.1, 0.3) & \perp & \perp \\ \perp & \perp & \perp & (0.6, 0.8] & (0.1, 0.2) \\ [0.8, 1.0) & \perp & \perp & \perp & \perp \\ \perp & \perp & \perp & [0.5, 0.7) & [0.1, 0.2] \\ (0.4, 0.8] & \perp & \perp & \perp & \perp \end{bmatrix}$$

To take advantage of the special properties of irreducible *PSXM* usage model, one must ensure that every state has a next state (the next state for the termination state is invocation) and exit transitions for each state have probability intervals, in which the maximal probabilities sum to 1. The numerical interval in entry (q_i, q_j) of the transition matrix represents the probability interval of a transition directly from q_i to q_j.

This matrix helps the test team to perform cost-benefit analysis on the planned testing based on the usage model. If the number of test cases required is too high, for example, cost reductions can be achieved by reducing the scope of the usage environment. Such scope reductions require careful consideration because the result is a constraint on inferences that can be made from testing.

Many transitions are associated with very small probabilities because they are followed infrequently. Grouping them together to form a single state would significantly simplify the resulting model and highlight the frequently used navigations. A simple lower-level usage model for this group can be obtained by linking this single grouped state to all those it represents to form a higher level usage model.

3. Generating Test Cases

Although much work exists in the validation and verification of traditional software, unfortunately, it is often hard to employ the traditional test theories and methodologies directly for web softwares because of their particularities and complexities. Many aspects regarding web test have not been sufficiently investigated yet, and many open questions still need to be addressed. In general, the behavioral (functional) conformance testing problem (also called fault detection problem) can be described as follows. Taking a specification and an IUT, if the latter is given as a black-box then only its input/output behavior can be observed, and consequently the testing process consists of comparing this observable behavior against the observable behavior of the specification. During testing, inputs are sent to, and outputs are received from, the IUT, then these outputs are compared with the expected outputs of the specification. From the execution of this process, a verdict about the conformance relation can be obtained, and any sequence that solves (at least partially) this problem is called a test sequence.

The standard *SXM*s test generation algorithm is based on the *W*-method introduced by [11] in the context of *FSM*s. A *SXM*-based test method [8, 12] enables us to derive a complete finite set of test cases that is proven to find all faults in the implementation. The outcome of the test generation algorithm is a finite set of input and expected output sequences. The inputs and expected outputs need to be mapped to concrete executable test cases that can be processed by a test engine, in order to interact with the system component under test and provide the results. The main point about *SXM* testing, the type of test generation technique discussed here, however, is not only to establish efficient test sets from a specification but also to relate the test process to a hierarchical decomposition of the implementation in order to make the management of the test process more convenient.

Definition 3.1 Given an implementation, *I*, of a specification, *S*, a *failure* occurs if for an input, *i*, the output produced by the implementation is unacceptable compared to the output produced by the specification. If the *S* and *I* are functions, then this can be written as $S(i) \neq I(i)$.

Definition 3.2 Any part of the system state that could lead to failures is a *fault*. For instance, a textual mistake in a program is a fault, which could lead to a number of individual failures. The fault is latent until these failures actually manifest themselves. If *I* is a faulty implementation of *S*, then this can be written as $S \neq I$.

Testing attempts to achieve correctness by detecting all the faults that are present in an implementation, so that they can be remove. In many cases, the act of designing a test case that would be affected by a particular fault means that the error leading to that fault is not made when constructing the implementation, or leads to the detection of that fault without having to actually execute the implementation and observe a failure.

It is possible to extract the test cases from the formal description. Then, it is possible to animate the model through the prototype with all the test cases and derive the expected sequences of output by the model. By feeding these test cases to the actual implementation, it is possible to compare the sequences of outputs with the sequences of outputs from the model in order to prove that the implementation is equivalent to the model.

Definition 3.3 Let $Z = (\sum, \Gamma, Q, M, \Phi, F, \Theta, I, T, m_0)$ be a *PSXM*. If $q, q' \in Q$, $\varphi \in \Phi$, $\rho \in \Theta$ and $q' \in F(q, \varphi, \rho)$, then φ is an *edge* from *q* to *q'*, which is denoted as $\varphi: q \rightarrow q'$.

Definition 3.4 In a *PSXM Z*, if $q, q' \in Q$ are such that $\exists q_1, \ldots\ldots, q_{n+1} \in Q$ with $q_1 = q$ and $q_{n+1} = q'$ so that $\varphi_1 : q_1 \rightarrow q_2$, $\varphi_2 : q_2 \rightarrow q_3, \ldots\ldots, \varphi_n : q_n \rightarrow q_{n+1}$, we say that we have a (test) path $p = (p_Q, p_\Phi)$ from *q* to *q'* of *Z* and denote as $p: q \rightarrow q'$, where $p_Q = q_1 \rightarrow q_2 \rightarrow \ldots\ldots \rightarrow q_n \rightarrow q_{n+1}$ is the sequence of states, called the *Q-path* (state path) of *p*, and $p_\Phi = \varphi_1 \ldots\ldots \varphi_n$ is the sequence of edges, called the Φ-*path* (function path) of *p*. Each Φ-*path* $p_\Phi = \varphi_1 \ldots\ldots \varphi_n$ gives rise to a (partial) function (or the path function) [*p*]: $M \times \sum^* \rightarrow \Gamma^* \times M$ defined by

$[p](m, s) = (g, m')$ if $\exists n \geq 0, \sigma_1, \ldots\ \ldots, \sigma_n \in \sum, \gamma_1, \ldots\ \ldots, \gamma_n \in \Gamma, m_1, \ldots\ldots, m_{n+1} \in M$ with $m_1 = m, m_{n+1} = m', s = \sigma_1, \ldots\ \ldots, \sigma_n$ and $g = \gamma_1, \ldots\ \ldots, \gamma_n$ so that $\varphi_i(m_i, \sigma_i) = (\gamma_i, m_{i+1})$, $\forall 1 \leq i \leq n$. The function corresponding to the empty path τ is defined by $[\tau](m, \tau) = (\tau, m), m \in M$.

The main reason for representing a path in this form is that for the same Φ-*path*, there could be more than one *Q-path* associated with it and vice versa. However, when the state sequence is irrelevant, it can be assumed that the path is given by the Φ-*path*. A path is a *complete* path, if and only if it starts from an initial state and ends in a final state, otherwise the path is said to be *partial*. A *loop* is a path that starts and ends in the same control state.

The proposed *PSXM* usage model can capture information about control flow, data flow, transaction processing and associated probabilistic usage, and criticality information. Test cases, which are sequences of events, can be generated from this information. To test web software, test cases must be generated. Test paths produced can be easily employed to construct test cases. A test case is one test path with user input values. So, a test path may be used to construct multiple test cases if only the test engineer provides different user input values. In the usage model, however, a test case is defined as any path from the source (initial state) to the sink (final state), i.e., a PSXM-based test path is regarded as an abstract test case.

Concretely, test cases can be produced by following the states and state transitions in *PSXM*s to select individual operations (states) and link them (transitions) together to form overall end-to-end operations. And the probability for the whole test path is the product of its individual transition probabilities. For example, there are at least three test paths from web page q_1 to q_5:

- $(q_1 \rightarrow q_2 \rightarrow q_4 \rightarrow q_5, \varphi_{submit2}\varphi_{browse}\varphi_{exit})$,
- $(q_1 \rightarrow q_2 \rightarrow q_4 \rightarrow q_4 \rightarrow q_5, \varphi_{submit2}\varphi_{browse}\varphi_{continue}\varphi_{exit})$ and
- $(q_1 \rightarrow q_2 \rightarrow q_5, \varphi_{submit2}\varphi_{logout})$.

The probability interval for the first test path is (0.6*0.6*0.1, 0.7*0.8*0.2], i.e., (0.036, 0.112], for the second one is (0.6*0.6*0.5*0.1, 0.7*0.8*0.7*0.2), i.e., (0.018, 0.0784), for the third one is (0.6*0.1, 0.7*0.2), i.e., (0.06, 0.14). From these three test paths, we can see that the maximal transition probability from q_1 to q_5 approximates 0.14 and the minimal one from q_1 to q_5 downward approximates 0.018.

Possible test cases with probabilities above specific thresholds can be generated to cover the operations that are frequently used. If the specific threshold is, for example, 0.12, then only the third path can be employed to generate test case because the maximal probability for the third path is greater than 0.12 and the former two maximal probabilities are less than 0.12.

For all practical purposes, the thresholds can be modified to control the numbers of test cases to be generated and executed. For example, we can start with a high threshold to test only the most frequently used operations and gradually lower the threshold to involve in more distinct situations and ensure the satisfactory performance or reliability for a wider variety of operations.

Moreover, we can also adapt test case allocation to cover critical parts or to compensate for increased complexity for complicated components of web softwares.

There is an important observation that can provide an approach to lessening the size of the *PSXM* usage model and this is as follows. All the paths from q to q' (in one change of configuration) have the same processing functions and they are independent of each other. This implies that all these processing functions belong to different components and each component participates with just one processing function. Therefore, any two different paths from q to q' produce the same sequence of processing functions for each of the components that participate in one change of configuration.

In order to know when to cease the testing, a stopping criterion is used. This should be defined before the testing starts. The stopping criterion is based on the reliability of the system and the coverage of the *PSXM* usage model. Hence, if the reliability is higher than expected and most probabilistic usage has also been tested, then the system can be released.

4. Related Work

Many web test challenges are discussed in [13], and a number of web test techniques have been already proposed [14-18], each of which has different origins and pursues different test goals for dealing with the unique characteristics of web softwares.

Andrews, et al. [14] illustrated an approach to modeling and testing web softwares based on FSMs after analyzing eight kinds of connections among web pages and software components of web softwares. They partitioned a web software into several functional clusters and logical pages, and tried to use hierarchical constrained FSMs to represent the logical pages and their navigations. However, the interactions and composition of components are not considered further.

Utting, et al. [15] provided an overview of model-based testing. Seven different dimensions define a taxonomy that allows the characterization of different approaches to model-based testing. It is intended to help with understanding benefits and limitations of model-based testing, understanding the approach used in a particular model-based testing tool, and understanding the issues involved in integrating model-based testing into a software development process. They classified several approaches embedded in existing model-based testing tools.

Miao, et al. [16] designed and implemented a model-based testing system for web softwares while the *FSM* is regarded as a formal testing model of web softwares under test. And this system integrates *Model Transformer*, *Test Purposes Analyzer*, *Test Sequences Generator*, *Visualization tools* for *FSM* and test sequences, *Test Execution Engine,* etc.

Qian [17] proposed a test case generation and optimization method for web software testing based on user sessions. He also expounded a testing approach [18] to web software using functional components.

Recently, several automated web testing tools have been developed to support web software testing [19-20]. These tools are integrated with specific web browsers to capture tests of user interactions with web softwares and create test scripts that can be replayed automatically to check the GUI components and verify the functionality of web softwares. However, these automated tools cannot provide structural and behavioral information of web applications to testers. Thus, it is still ad hoc for testers to design test cases effectively, especially if the design documents are incomplete or missing.

SXM-based testing has been developed in various directions. Moreover, numerous empirical studies have been carried out to establish if the approach is actually practical. Thus, a number of systems have been specified as *SXM*s and those tests produced by the approach are utilized to validate the implementations.

The main advantages of using *SXM* model for representing systems is that it is flexible and allows the integration of control and data processing. This formalism has been used to specify systems in different areas and several testing techniques have been developed.

Bogdanov, et al. [21] described the *X-machine* testing method and its use for testing of different types of systems, both in terms of theory and practical outcomes. They surveyed the extensions of the *X-machine* testing method for testing of functions together with testing of a transition diagram, equivalence testing of a non-deterministic implementation against a non-deterministic specification, conformance testing of a deterministic implementation against a non-deterministic specification and equivalence testing of a system of concurrently executing and communicating *X-machine*s, against a specification.

Ipate and Gheorghe [22] presented the complete non-deterministic *SXM* testing method to generalize the non-deterministic *SXM* integration testing method. It no longer requires implementations of the processing relations to be proved correct before integration testing can take place. Instead, the testing of processing relations is performed along with the integration testing. The authors also showed how a *SXM* model of a *P* system can be obtained and how the non-deterministic *SXM* testing approach can be applied to generate conformance test sets for the *P* system.

Ipate and Holcombe [23] provided a new variant of the *SXM*-based testing method that no longer depends on the size of a *controllable* model of the implementation under test. In data processing-oriented applications, the new method can drastically reduce the size of the test suite produced at the expense of a (possibly) more complex generation process.

Merayo, et al. [5] stated a formal testing framework for systems where timeouts are critical. The model introduced for specifying the systems is a suitable extension of the classical concept of *SXM*. They introduced a notion of test that can delay the execution of the implementation and also proposed an algorithm to derive sound and complete test sets.

The animation process can be readily supported by existing tools. *X-System* [9] is a Prolog-based tool supporting the animation of *SXM* models, while a Java-based graphical user interface on top of *X-System* is also available. In addition to animation, model checking techniques can be employed on the *SXM* model to check for desirable or undesirable properties specified with temporal logic formulae. Research on *X-machine*s already offers a model checking logic, called *XmCTL*, which extends Computation Tree Logic (*CTL*) with memory quantifiers in order to facilitate model checking of temporal properties in *X-machine* models [24].

5. Discussions, Conclusions and Perspectives

5.1. Discussions

It implies many benefits to producing test cases based on a formal model, whether it is a specification or an implementation. The benefits arise from the ability to precisely depict and reason about potential faults. Particularly, it means that tests can be applied uniformly with greater confidence in their fault detecting capability and with the possibility of full automation.

The *SXM* model has become increasingly important over the past decade, because it has full recursive power (any recursive algorithm can be articulated), and yet its control theory is essentially identical to that of finite state transducers. It is consequently easy to extend standard transducer test strategies to the *SXM* with only minimal changes, and so obtain a test strategy for general recursive algorithms.

As we know, a *SXM* is a general computational machine that can model the non-trivial data structures as a typed memory tuple and the dynamic part of a system by employing transitions, which are not labeled with simple inputs but with functions that operate on inputs and memory values. The *SXM* formal method is valuable to software engineers since it is rather intuitive, while at the same time formal descriptions of data types and functions can be written in any known mathematical notation. These differences allow the *SXM*s to be more expressive and flexible than an *FSM*. In addition, a set of *SXM*s can be viewed as components, which communicate with each other in order to specify larger systems.

A very powerful attribute is that the *SXM*-based testing method provides automatically, an effective mechanism for managing the test process and aligning it to modern software and hardware engineering techniques such as component-based design whereby *basic functions* can be reused, and the test problem becomes one of testing the integration of a number of dependable components. Usually, this method generates a test set from a deterministic *SXM* specification, providing that the system components (i.e. the processing functions) are implemented correctly. Therefore, it is assumed that the implementation is a deterministic *SXM* having the same processing functions as the specification. In fact, it is proved that only if the specification and the implementation are behaviorally equivalent, the test set produces identical results when applied to both of them. Otherwise, it is guaranteed that it will reveal the faults in the implementation.

The primary problem in *SXM*-based testing is that it is difficult to ensure that the function φ ($\in \Phi$) is error-free. This is due to the fact that there is no clear distinction between control (represented by the transition diagram) and data transformations (performed by functions), and hence functions cannot be tested separately from the testing of the state transition diagram.

5.2. Conclusions

At present, there are no systematic method and tool that are employed to test web softwares efficiently. The improved traditional methods or a new method appropriate for web software testing are desired urgently for all the characteristics of web softwares. Since the current testing methods depend primarily on the testers' intuition and experience, the testing of web softwares is regarded as a time-consuming, tiresome and expensive drudgery. Therefore, a new methodology for web testing is required imminently to automate the testing.

The primary benefit of statistical testing is that it allows the use of statistical inference techniques to compute probabilistic aspects of the testing process. It is not to cover the code, but to validate the future usage of the system in order to guarantee a use without failure of the released system.

This way, the presented *PSXM* usage model spans the chasm between requirements engineering and reliability engineering. It is characterized by the following features:

- It uses a formal method, which employs *PSXM* as a mathematical tool convenient for formal deduction.
- It is usage-oriented and specification-based, thus performing black-box testing.
- It aims at statistical reliability testing rather than fault detection only. Test cases can be generated by a random walk starting in the source and terminating upon reaching the sink constrained by the transition probabilities of the *PSXM* usage model.

In testing, we can choose to run first the test set and test case with higher priority, and then those with lower priority, or run the test sets concurrently. Once some test case in some test set is executed and test requirements are satisfied, the remainder test sets and test cases are no longer executed yet. This lessens the executing time greatly. In this case, there is an executing precedence relationship among different test sets and test cases in each test set. The test cases are grouped, therefore, the testers can choose to run several test sets first to find some types of errors in higher probability and in less time. Thus, our approach can be easily extended to be much sound and flexible.

5.3. Perspectives

The description of the *PSXM* usage model can provide a powerful way to generate test cases for research. However, a great deal still needs to be done. Verification and testing should complement each other with testing providing confidence in the correctness of the assumptions made in verification. As our further research, the study on the combination of verification and testing applied to the proposed *PSXM* usage model is of great interest and challenge.

Acknowledgements

This work was partially supported by National Natural Science Foundation of China (NSFC) under grant nos. 61262010, 61163007, 61462030, 61262034 and 61272007, China Postdoctoral Science Foundation Funded Project under grant no. 2014M551855, Jiangxi Provincial Natural Science Foundation of China under grant no. 20132BAB201036, Jiangxi Provincial Postdoctoral Scientific Research Preferred Funded Project of China under grant no. 2013KY40 ([2013]227), and Scientific Research Key Program of Jiangxi Provincial Education Department of China under grant no. GJJ13307.

References

[1] Z. Li, N. Alaeddine and J. Tian, "Multi-Faceted Quality and Defect Measurement for Web Software and Source Contents", Journal of Systems and Software, vol. 83, no. 1, **(2009)**, pp. 18-28.
[2] T. Huynh and J. Miller, "Another Viewpoint on 'Evaluating Web Software Reliability Based on Workload and Failure Data Extracted from Server Logs", Empirical Software Engineering, vol. 14, no. 4, **(2009)**, pp. 371-396.
[3] S.-Y. Wu, "Establishment of Web Software Cost Estimation Model", Master's Thesis, Ming Chuan University, **(2010)**.
[4] B. Zhao, "Software Testing Technology Classic Course", Beijing: Science Press, China, **(2007)** (in Chinese).
[5] M. G. Merayo, R. M. Hierons and M. Nunez, "Extending Stream X-machines to Specify and Test Systems With Timeouts", Proceedings of the 6th IEEE International Conference on Software Engineering and Formal Methods, IEEE CS, **(2008)**.
[6] G. Laycock, "The Theory and Practice of Specification-Based Software Testing", PhD thesis, Dept of Computer Science, Sheffield University, UK, **(1993)**.
[7] S. Eilenberg, "Automata, Languages and Machines", Academic Press, vol. A, **(1974)**.
[8] M. Holcombe and F. Ipate, "Correct Systems: Building Business Process Solutions", S. Verlag, **(1998)**.
[9] E. Kapeti and P. Kefalas, "A Design Language and Tool for X-machine Specification", Advances in Informatics, Fotiadis D. and Nikolopoulos S. (Eds), World Scientific Publishing Company, **(2000)**, pp. 134-145.
[10] P. Kefalas, G. Eleftherakis and A. Sotiriadou, "Developing Tools for Formal Methods", Proceedings of the 9th Panhellenic Conference in Informatics (PCI), **(2003)**.
[11] T. S. Chow, "Testing Software Design Modelled by Finite State Machines", IEEE Transactions on Software Engineering, vol. 4, no. 3, **(1978)**, pp. 178–187.
[12] F. Ipate and M. Holcombe, "An Integration Testing Method That Is Proven to Find All Faults", International Journal of Computer Mathematics, vol. 63, **(1997)**, pp.159-178.
[13] G. A. Stout, "Testing a Website: Best Practices", A Whitepaper. http://www.reveregroup.com, **(2014)**.
[14] A. Andrews, J. Offutt and R. Alexander, "Testing Web Applications by Modeling with FSMs", Software Systems and Modeling, vol. 4, no. 3, **(2005)**, pp. 326-345.
[15] M. Utting, A. Pretschner and B. Legeard, "A Taxonomy of Model-based Testing Approaches", Software Testing, Verification and Reliability, **(2011)**.
[16] H. K. Miao, S. B. Chen and H. W. Zeng, "Model-Based Testing for Web Applications", Chinese Journal of Computers (in Chinese with English abstract), vol. 34, no. 6, **(2011)**, pp. 1012-1028.
[17] Z. S. Qian, "Test Case Generation and Optimization for User Session-based Web Application Testing", Journal of Computers, vol. 5, no. 11, **(2010)**, pp. 1655-1662.
[18] Z. S. Qian, "Towards Testing Web Applications Using Functional Components", Journal of software, vol. 6, no. 4, **(2011)**, pp. 740-745.
[19] Mercury Interactive, http://www.merc-int.com, **(2014)**.
[20] E. Miller, "WebSite Testing", http://www.soft.com/Products/Web/Technology/website.testing.html, **(2014)**.
[21] K. Bogdanov, M. Holcombe, F. Ipate, L. Seed and S. Vanak, "Testing Methods for X-machines: a Review", Formal Aspects of Computing, vol. 18, no. 1, **(2006)**, pp. 3-30.
[22] F. Ipate and M. Gheorghe, "Testing Non-deterministic Stream X-machine Models and P systems", Electronic Notes in Theoretical Computer Science, vol. 227, **(2009)**, pp. 113-126.
[23] F. Ipate and M. Holcombe, "Testing Data Processing-oriented Systems From Stream X-machine Models", Theoretical Computer Science, vol. 403, no. 2-3, **(2008)**, pp. 176-191.
[24] G. Eleftherakis, P. Kefalas and A. Sotiriadou, "XmCTL: Extending Temporal Logic to Facilitate Formal Verification of X-machines", Analele Universitatii Bucuresti, Matematica-Informatica vol. 50, **(2001)**, pp. 79-95.

Author

Qian Zhongsheng. He was born in Yingtan city, Jiangxi province, China in 1977. He received his M.S. degree from Nanchang University of China in 2002, and Ph.D. in Computer Science at Shanghai University of China in 2008 respectively.

His current research interests include software engineering, web software testing and social network, etc.

Dr. Qian has also been with the School of Information Technology at Jiangxi University of Finance & Economics of China since 2002.