Software Reliability Test Based on Markov Usage Model

Kuanjiu Zhou
School of Software, Dalian University of Technology, Dalian, China
Email: zhoukj@dlut.edu.cn

Xiaolong Wang, Gang Hou*, Jie Wang and Shanbin Ai
Email: {wxl_dut, hg.dut, wangjie1003, aishanbin}@163.com

Abstract—The reliability test for embedded software system is very difficult due to its complex structure and large scale. Markov usage model which describes the software usage with Markov process is widely used for statistical test. Software statistical test based on Markov usage model is an effective approach to the generation of test cases with high efficiency and the evaluation of software reliability in a quantitative way. This paper mainly focuses on the generation of Markov usage model of software system and the method of software reliability test based on it. Firstly, a method to build Markov usage model based on improved state transition matrix (STM), which is a table-based modeling language, is proposed. Then a software reliability test method including test case generation and test adequacy determination based on Markov usage model is presented. An improved Kullback discriminant is selected as the judgment criteria of convergence from the test chain to the usage chain in order to measure whether the testing process is sufficient. Finally, a Markov test cases generation tool MTCG is developed which implements the methods put forward in this paper. Experimental verification of test adequacy and efficiency is made through a comparison between the Markov usage model-based method and a completely random test method, the results of which show that software reliability test based on Markov usage model is approving, high-efficient and promising.

Index Terms—Markov Usage Model, Software Reliability Test, Test Case Generation, Test Adequacy

I. INTRODUCTION

The reliability of embedded software system attracts more and more attention since it is widely applied in aerospace, military, medical, communications, nuclear reactions, and industrial control areas. Due to the large scale and complex structure of embedded software system, once the software system fails, inestimable consequences would be caused, such as loss of life and property, severe damage to the natural environment. For example, in June 1996, the European Space Agency's Ariane 5 launch vehicle exploded catastrophically after 37 seconds took-off because of software failures. In the mid-1980s, the United States Therac-25 radiation therapy went out of order in 2nd treatment mode, resulting in a severe medical malpractice of two deaths and several injuries, because its security system had not been tested sufficiently after its interlock device was replaced. Most of those accidents are owing to inadequate software testing, which cannot guarantee the software reliability effectively.

Different from traditional software testing, software reliability test requires the software usage model to generate test cases. Software reliability can be evaluated according to the statistical results of software testing. There are three basic problems of software reliability test, the reliability test model, test adequacy and test case generation [1-3].

Software testing method based on usage model can effectively reduce the number of test cases, save testing costs and improve software reliability, since it is much closer to the actual usage of software system than the completely random testing method. Markov usage model is a commonly used model in software reliability test, which is widely used in the fields such as information theory, automatic control, communication techniques, computer sciences and genetics [4].

Much research has been done on Markov usage model. In 1994, Whittaker and Thomason applied Markov chain to software reliability test for the first time [5]. Wang Ji and Chen Huowang proposed a method of Markov chain usage model based on UML use case diagram and state diagrams which provides formal description base for testing UML-based software [6]. Gao Haichang used the Kullback discriminant as the convergence judgment criteria of the test-chain to usage-chain, and testified the inevitability of the convergence theoretically [7]. Murali Krishna proposed a new semi-Markov process-based model to compute the network parameters such as saturation throughput for the IEEE 802.11 Distributed Coordination Function (DCF) employing the Binary
Exponential Backoff (BEB) [8]. He Yiming and Pu Yingxia applied Markov chains and Spatial Markov chain to investigate the spatial and temporal characteristics of industrial structure level [9]. Liu YS proposed the CSP (checkpoint-based spare processor) model which is an improvement of the traditional SP (spare processor) model to solve the fault-tolerant problem of SP [10]. Li Mingshu proposed an adaptive multiagent negotiation model for software process modeling, namely AMNM-PA, which is based on non-stationary finite-horizon Markov decision processes and uses the model-independent Q learning algorithm to choose negotiation strategies [11]. Mo YC and Yang XZ demonstrated how to analyze mission reliability of the generalized PMS with random phase duration and Markov regenerative intra phase processes to attack the weak points of the state-of-the-art [12]. Liu Y and Ma ZY studied the performances of various congestion control strategies through building accurate Markov chain model-intensive Q learning algorithm to choose negotiation strategies [11]. Mo YC and Yang XZ demonstrated how to analyze mission reliability of the generalized PMS with random phase duration and Markov regenerative intra phase processes to attack the weak points of the state-of-the-art [12]. Liu Y and Ma ZY studied the performances of various congestion control strategies through building accurate Markov chain model-intensive Q learning algorithm to choose negotiation strategies [11].

DG presented a method based on Hidden Markov model for the quantitative evaluation of reliability [13]. Kong Yingxia applied Markov chains and Spatial Markov chain based on Markov usage model, including test case mining applications [18]. In addition, Tian Xinguang and Zhang Youzhi integrated the concepts presented above, the Markov chain transfers to one of the states in \( \{a_i, a_{i+1}, \ldots\} \) at the moment \( n + m \) from the state \( a_i \) at the moment \( m \) inevitably, the following equality can be discovered:

\[
\sum_{j=0}^{m} P_j(m, m + n) = 1, (i = 1, 2, \ldots)
\]

A matrix constituted of transition probability is called the transition probability matrix of a Markov chain. The transition probability matrix is of great use in the generation of test cases and in the determination of test adequacy. Based on (4), we can obviously conclude that the summation of the elements value in each row is 1 in a transition probability matrix. Our major work in this paper focuses on the construction and verification of Markov usage model, and how to generate test cases as few as possible based on the usage model in the premise of ensuring test adequacy in order to improve the efficiency of embedded software reliability test.

This paper is structured as follows. Section II presents an introduction of Markov usage model and proposes a modeling method for Markov usage model based on STM. In Section III, we put forward a software testing method based on Markov usage model, including test case generation and test adequacy determination. In Section IV, we carry out an experiment with an independently developed test case generation tool MTCG and make verification on the efficiency and adequacy of software reliability test based on Markov usage model. The conclusions are drawn in Section V.

II. MARKOV USAGE MODEL

A. Introduction of Markov Usage Model

Markov usage model is a classic software usage model, which describes the software usage with Markov process. In probability and statistics theory, if the state of a process or a system at the moment \( t_0 \) is known and its state at the moment \( t \) has no relationship with any other states before \( t_0 \) ( \( t > t_0 \) ), we can conclude that the process is a Markov process and the process or the system has Markov property, which is also called memoryless property.

The Markov process can be described with a distribution function. Assume that a stochastic process \( X(t) \) ( \( t \in T \) ) is a Markov process and has a state space \( I \). For \( t_i \leq t_i < t_{i+1} \leq t_{i+2} \ldots \leq t_n \) the conditional probability distribution of \( X(t_i) \) under the conditions that \( X(t_i) = x_i \) \( (x_i \in I, i = 1, 2, \ldots, n-1) \) equals its conditional probability distribution under the condition that \( X(t_{i-1}) = x_{i-1} \). That is to say, for \( x_i \in \mathbb{R}, \)

\[
P_i[X(t_i) \leq x_i | X(t_i) = x_i, X(t_{i-1}) = x_{i-1}]
\]

(1)

The Markov chain is the Markov process with discrete time and states, which can be denoted by \( \{X_n = X(n), n=0, 1, 2, \ldots\} \) ( \( a_i \in \mathbb{R} \)), the Markov property can also be described with the conditional probability distribution as follows:

\[
P(X_{m+n} = a_j | X_m = a_i, X_{m+n} = a_j, \ldots X_{m+n} = a_n) = P(X_{m+n} = a_j | X_m = a_i)
\]

(2)

Here, \( n, r \in \mathbb{N} \), \( 0 \leq t_1 < t_2 \ldots < t_r < t_m \).

The conditional probability \( P_j \) denotes the transition probability of the Markov chain from state \( a_i \) at the moment \( m \) to state \( a_j \) at the moment \( m+n \). The definition of \( P_j \) can be described as follows:

\[
P_j(m, m + n) = P[X_{m+n} = a_j | X_m = a_i]
\]

(3)

As the Markov chain transfers to one of the states in \( \{a_i, a_{i+1}, \ldots\} \) at the moment \( m+n \) from the state \( a_i \) at the moment \( m \) inevitably, the following equality can be discovered:

\[
\sum_{j=0}^{m} P_j(m, m + n) = 1, (i = 1, 2, \ldots)
\]

A matrix constituted of transition probability is called the transition probability matrix of a Markov chain. The transition probability matrix is of great use in the generation of test cases and in the determination of test adequacy. Based on (4), we can obviously conclude that the summation of the elements value in each row is 1 in a transition probability matrix.

Integrated the concepts presented above, the Markov usage model is a sequence of states and transitions with the key characteristic that the appearance probability of state \( s_i \) depends only on the previous state and is independent of any other history states.

Fig. 1 gives an example of Markov usage model, \{START, A, B, C, EXIT\} is the state space, in which START denotes the beginning state of the software usage and EXIT denotes the ending state. \{a, (b, 0.5), (c, 0.5), (d, 0.75), (e, 0.25), (f, 0.5), (g, 0.25), (h, 0.25)\} is the transition set of the usage model and the decimals
denote the transition probability. Especially, for an embedded software system, the transitions or events in the transition set can be used to denote interrupts or messages.

![A UML state machine diagram showing transitions and states labeled with probabilities and actions.](image)

Figure 1. An example of Markov usage model.

**B. Modeling Method Based on STM**

One of the key problems in software testing based on Markov usage model is how to construct the model in the requirements modeling. Nowadays, some useful research in the Markov modeling has been done in Reference [1-2, 5-7]. In this paper, we present a method of modeling Markov usage model based on State Transition Matrix (STM) [23-24]. STM is a table-based modeling language for describing behavior of distributed systems [25-26]. Kong WeiQiang has given a detailed description of STM [27], and uses STM to model checking for embedded software. In this paper, we improve the form of STM by adding the probability of events, in order to suit for the software. In this paper, we improve the form of STM by adding the probability of events, in order to suit for the

markov usage model based on state transition matrix (STM) [23-24]. STM is a table-based modeling language for describing behavior of distributed systems [25-26]. Kong WeiQiang has given a detailed description of STM [27], and uses STM to model checking for embedded software. In this paper, we improve the form of STM by adding the probability of events, in order to suit for the software. In this paper, we improve the form of STM by adding the probability of events, in order to suit for the

figure 2. An example of state transition matrix(STM).

We firstly describe an action language $L$ that is needed for defining the structure of STMs. $L$ is chosen to be a simple subset of ANSI C language and thus the syntax and semantics follow the conventions of C programming language. Type system of $L$ consists of Boolean, Integer, and Real. Supported expressions of $L$ are (A) Boolean literals true and false, Integer literals and Real literals, (B) Variable identifiers, and (C) infix expressions $leq expr op rightexpr$, where $op$ can be one of $+, -, *, /, %, \&\&, ||$, $<, \leq, =, \neq$, or $!=$, with the semantics of ANSI C. Supported statements of $L$ are (A) assignments of the form $lhs = rhs$, and (B) if statements of the form if condition $[statement1]$ else $[statement2]$.

Assuming this action language $L$, the structure of a STM $H$ is a tuple $<S,E,P,C>$, where

- $S$ is a finite set of state. Each state $s \in S$ is associated with a (unique) index number denoted by $index(s) \in Nat$. During execution of $H$, only one state denoted by $active(H)$ is active. Initially, the active state is $s$ where $index(s) = 0$.
- $E$ is a finite set of events consisting of external events $E_{\text{external}}$ and internal events $E_{\text{internal}}$, where $E_{\text{external}} \cap E_{\text{internal}} = \emptyset$. An event $e \in E_{\text{external}}$ is represented by a Boolean variable of $L$ (whose name is prefixed with a lower-case “x” by convention), and an event $e_1 \in E_{\text{internal}}$ is represented by a Boolean expression (possibly a Boolean variable) of $L$. Each event is associated with a (unique) index number denoted by $index(e) \in Nat$.
- $P$ is a finite set of transition probability. $P_{ij} \in P$ means the probability of event $E_i$ occurring, while in the state $S_j$. Here, the value of $P_{ij}$ is limited by two types of constraint, one is structure constraint, and the other is usage constraint. Structure constraint reflects the basic properties of Markov usage chain, mainly including (A) $0.001 < P_{ij} < 0.999$ and (B) the summation of $P_{ij}$, $P_{ij}, ...$, $P_{nn}$ is 1. Structure constraint is fitted for all software system. Usage constraint reflects the actual usage of software system, including (A) linear constraints such as $P_{2,3} = P_{3,5}$, $P_{5,3} = P_{3,1} + P_{1,3} + P_{3,5}$, and (B) interval constraints such as $0.25 < P_{1,3} < 0.35$, $0.5 < P_{1,5} < 0.85$. Usage constraint can be obtained from requirements analysis and usage statistics of software system. Based on these constraints above, all the transition probability in $P$ can be figured out by solving the equation constituted of inequality constraints.

- $C$ is a finite set of cells consisting of Normal cells $C_{\text{normal}}$, Ignore cells $C_{\text{ignore}}$, and Invalid cells $C_{\text{invalid}}$. Each normal cell $c_n \in C_{\text{normal}}$ is a tuple $< s, e, u, a, s' > \in S \times E \times \{ \text{null} \} \times \{ \text{null} \} \times S$. We define $source(c_n) = s$, $event(c_n) = e$, $guards(c_n) = u$, $actions(c_n) = a$, and $target(c_n) = s'$. Each Ignore cell is a tuple $< s, e, >$. Each Invalid cell is a tuple $< s, < e, >>$. Functions $source$ and $event$ are also defined for Ignore and Invalid cells $c_i \in C_{\text{ignore}} \cup C_{\text{invalid}}$ as for $c_n$, but guards, actions, and target are not.

Events of a STM and a Guards of a Normal cell are expressed as $L$ expressions of type Boolean, and actions of a Normal cell are a list of $L$ statements. A cell $e$ of a STM $H$, which is pinpointed by its index numbers ($index(source(e))$, $index(event(e))$) together with its guards if $guards(e) \neq \text{null}$, specifies the behavior of $H$ (under condition of $guards(e)$ if available) when event...
event(c) is dispatched while \( H \) is in state source(c) (i.e., \( \text{active}(H) = \text{source}(c) \)). If \( c \in C_{\text{normals}} \), actions(c) will be executed atomically and after that, \( H \) moves to state target(c). If \( c \in C_{\text{ignore}} \), denoted in a STM table by the symbol "\( \times \)”, nothing changes. If \( c \in C_{\text{invalid}} \), denoted in a STM table by the symbol "\( \times \)”, an error occurs. Informally, an Ignore cell means that the dispatch of an event in a state is ignored, and an Invalid cell means that the dispatch of an event in a state is never possible.

### III. SOFTWARE TESTING METHODS BASED ON MARKOV USAGE MODEL

The Markov usage model constructed with the methods given above provides an essential model for the reliability testing of embedded software. The method of test case generation based on Markov usage model is the key problem in reliability test of embedded software and the most important job we focus on in this paper.

#### A. Algorithm of the Generation of Test Cases

In the generation of test cases for embedded software based on Markov usage model, a test case is a sequence of states or transitions passing through the Markov test chain from the beginning state to the ending state. The Markov test chain is formed from the usage model as follows:

Firstly, the test chain is built based on the structure of the usage chain and each transition probability of the edges is replaced to a counter with an initial value 0. Then, as the testing process goes forward, the counter of the edge increases by 1 if a test case goes through it. Finally, the counter of each edge is converted to transition probability based on the statistical results.

In this paper, we present an algorithm of test case generation based on Markov usage model in software reliability test as follows:

**Step 1**: Let \( st_{\text{curr}} \) denote the current state and set the value of \( st_{\text{curr}} = \text{START} \).

**Step 2**: Generate a uniform random decimal \( \text{rnd} \) between 0 and 1 at the state \( st_{\text{curr}} \) and select a transition with the strategy in Step 3.

**Step 3**: Based on all the transitions \( \{(t_i, p_k)\} (k=1, 2, ..., m) \) from the state \( st_{\text{curr}} \) to the state \( st_{\text{next}} \) in the usage model, divide the interval \([0, 1]\) into \( m \) intervals \([I_1, I_2, ..., I_m]\). Here \( I_1 = [0, p_1) \), \( I_2 = [p_1, p_1+p_2) \), ..., \( I_m = [p_1+p_2+...+p_{m-1}, 1) \). If \( \text{rnd} \in I_k \), record the transition \( t_k \) and set \( st_{\text{curr}} = st_{\text{next}} \).

**Step 4**: Repeat Step 2 and Step 3 until the value of \( st_{\text{curr}} \) equals \( \text{EXIT} \) and all the transitions recorded in Step 3 form a test case.

**Step 5**: Repeat all the steps above to generate enough test cases until the testing process is sufficient.

In the algorithm given above, test cases are generated from uniform random decimals referenced to transition probability of Markov usage model. This algorithm not only satisfies the randomness of the test case generation process, but also takes the transition statistical probability of software usage into account. That is why we can generate test cases more corresponded to the actual usage of software system with a smaller quantity. Table I shows an example of a test case generated based on the Markov usage model in Fig. 1.

<table>
<thead>
<tr>
<th>Order</th>
<th>Transition/Event</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>c</td>
<td>C</td>
</tr>
<tr>
<td>3</td>
<td>g</td>
<td>B</td>
</tr>
<tr>
<td>4</td>
<td>e</td>
<td>C</td>
</tr>
<tr>
<td>5</td>
<td>f</td>
<td>C</td>
</tr>
<tr>
<td>6</td>
<td>h</td>
<td>EXIT</td>
</tr>
</tbody>
</table>

#### B. Feasibility of Test Case Generation Algorithm

In software testing based on Markov usage model, the key point to ensure the feasibility of test case generation algorithm is to generate random paths in which \( P_{ij} \) is equal to \( P_i \). \( P_{ij} \) is the output edge transition probability of state \( S \) in the test chain, and \( a_{ij} \) is the corresponding edge to \( a_i \) in the usage chain.

Assume that the state \( S \) has \( k \) output edges, such as \( a_1, a_2, ..., a_i, P_i, P_2, ..., P_k \) are their corresponding probability. We have the following equality

\[
\sum_{i=1}^{k} P_i = 1
\]

(5)

Assume that the number of visiting state \( S \) in the test chain is \( n \) and the number of output edge \( a_i (1 \leq i \leq k) \) is \( m \), corresponding to the Bernoulli theorem we have the conclusion that

\[
\lim_{n \to \infty} \frac{m}{n} = P_j
\]

(6)

According to the above formula, it can be ensured than the transition probability \( a_{ij} \) in the test chain is equal to the access probability \( P_{ij} \) in the usage chain when \( n \) is large enough as long as the random decimal generation obeys the 0–1 uniform random distribution.

\[
P_{ij} = \int_{0}^{1} dx \sum_{k=1}^{m} P_k - \sum_{k=1}^{m} P_k - P_j = P_j
\]

(7)

It is obviously that \( P_{ij} \) is always equal to \( P_j \) when \( n \) is large enough as long as the random decimals generated for the path selection obey the 0–1 uniform random distribution.

#### C. Test Adequacy Determination Criteria

Test adequacy determination criteria become an essential problem that must be addressed in software reliability test. The test adequacy determination criteria based on Markov usage model is a model comparison criteria which comparing the usage environment (usage chain) and test environment (test chain). The test can be considered sufficient until the test chain impend over the usage chain.

The test chain \( U \) and the usage chain \( T \) can be seen as two different descriptions of Markov process in software...
reliability test theory based on Markov usage model, so the Discriminant value can be applied to determine the adequacy of software testing. Discriminant value, also known as Kullback discriminant, is the mathematical expectation of convergence value of two random processes. For simplicity, discriminant value is denoted by $D(U, T)$, where $U$ represents the usage chain, $T$ represents the test chain. The value of $D(U, T)$ is closer to 0, the test chain $T$ is more similar to the usage chain $U$. The testing process is considered sufficient and can be stopped when the discriminant value is less than the expected threshold value. Discriminant value can be calculated accurately as follows:

$$D(U, T) = \lim_{n \to \infty} \frac{1}{n} \sum_{i=1}^{n} \log \left( \frac{Pr(X_{\lambda}, X_{\lambda+1}, ..., X_{\lambda+n} | U)}{Pr(X_{\lambda}, X_{\lambda+1}, ..., X_{\lambda+n} | T)} \right)$$

(8)

$Pr(X_{\lambda}, X_{\lambda+1}, ..., X_{\lambda+n} | \lambda)$ represents the circumstances of executing $n$ set of test cases under the condition of probability distribution with parameter $\lambda$. The $D(U, T)$ cannot be obtained directly by calculating the usage chain $U$ and the test chain $T$ and the test. It’s equivalent formula as follows:

$$D(U, T) = \sum_{i=1}^{n} \pi_i \sum_{j=1}^{n} u_{ij} \log \frac{u_{ij}}{t_{ij}}$$

(9)

The $u_{ij}$ and $t_{ij}$ represent the transition probability from state $i$ to state $j$ in usage chain and test chain respectively. $\pi_i$ represent the long-run occupation ratio of state $i$. Clearly, the state's occupation ratio in the long-run is equal to the sum of products of adjacent state's occupation ratio and the transition probability from the adjacent state to the state. And the sum of all state’s transition probability in the long-run is equal to one. The calculate method as follows:

$$\pi_j = \sum_{i=1}^{n} \pi_i P_{ij}$$

(10)

$$\sum_{i=1}^{n} \pi_i = 1$$

(11)

The discriminant value of $D(U, T)$ is not always calculable according to (9). If the test chain does not cover all edge of the usage chain-chain, the value of $t_{ij}$ is zero and is not calculable. Therefore, only when the test chain covers all the edge, the $D(U, T)$ is feasible and meaningful.

We adopt an improved Kullback discriminant, denoted $K(U, T)$, to be criteria for judging the adequacy of software reliability test. The $K(U, T)$, as follows, is a variant calculation method to $D(U, T)$ which is always calculable after modified.

$$K(U, T) = \sum_{i=1}^{n} \pi_i \sum_{j=1}^{n} u_{ij} \log \frac{u_{ij}}{e^{-e(\text{sgn}(t_{ij})) + t_{ij}}}$$

(12)

Here, $e$ is a sufficiently small positive decimal. $\text{sgn} (x)$ represent a function: When $x = 0$, $\text{sgn} (x) = 0$; when $x < 0$, $\text{sgn} (x) = 1$; when $x > 0$, $\text{sgn} (x) = 1$. Obviously, when the test chain $T$ does not cover all the edges, $K(U, T)$ is calculable as usual. When the test chain $T$ covers all edges, $K(U, T)$ is equal to $D(U, T)$.

$K(U, T)$

The closer is the test chain $T$ to the usage chain $U$, the nearer to zero is the $K(U, T)$. The convergence threshold can be set properly and acceptably according to the parameters of different software systems. When the value of $K(U, T)$ is less than the threshold, the test adequacy can be considered reached and the testing process can be stopped.

IV. EXPERIMENTS AND VERIFICATION

In this paper, a generation tool of test cases based on Markov usage model, named MTCG (Markov Test Cases Generator), was successfully developed and implemented in C# language with Microsoft Visual Studio 2008 integrated development environment and .NET Framework 3.5 framework. This test case generation tool MTCG enable software testers to customize and verify a Markov usage model of software system based on STM and then automatically draw a graph for the usage model. However, the main function of MTCG is to generate test cases based on the usage model customized above automatically and get the statistical information in testing process in order to evaluate and improve the reliability of the software. Using MTCG, we carried out an experiment of software testing based on the usage model shown in Fig. 1 and analyzed the adequacy of the testing process and the efficiency of the testing method proposed in this paper.

A. Adequacy Verification of Software Reliability Test Process

Based on the improved Kullback discriminant proposed in section III, an experiment was carried out with $\epsilon=0.00001$ and $K(U, T)$ threshold = 0.001 to test and verify the adequacy of the testing process. Table II shows the statistical results in the experiment, including the node coverage, edge coverage and the value of $K(U, T)$ with different number of test cases.

<table>
<thead>
<tr>
<th>No. of Cases</th>
<th>Node Coverage</th>
<th>Edge Coverage</th>
<th>$K(U, T)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0.8000</td>
<td>0.5000</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1.0000</td>
<td>0.8750</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>1.0000</td>
<td>1.0000</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>1.0000</td>
<td>1.0000</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>1.0000</td>
<td>1.0000</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td>1.0000</td>
<td>1.0000</td>
</tr>
<tr>
<td>7</td>
<td>20</td>
<td>1.0000</td>
<td>1.0000</td>
</tr>
<tr>
<td>8</td>
<td>50</td>
<td>1.0000</td>
<td>1.0000</td>
</tr>
<tr>
<td>9</td>
<td>100</td>
<td>1.0000</td>
<td>1.0000</td>
</tr>
<tr>
<td>10</td>
<td>200</td>
<td>1.0000</td>
<td>1.0000</td>
</tr>
<tr>
<td>11</td>
<td>300</td>
<td>1.0000</td>
<td>1.0000</td>
</tr>
<tr>
<td>12</td>
<td>391</td>
<td>1.0000</td>
<td>1.0000</td>
</tr>
</tbody>
</table>
The first two rows of Table II show that the value of $K(U, T)$ also can be calculated with the use of improved Kullback discriminant, even under the condition that the test cases failed to cover all the migration edges in usage model. This result coincides well with the theory proposed in section III and also indicates that the calculation method of $K(U, T)$ with the improved Kullback discriminant is approving.

Statistical data in Table II shows that the convergence value $K(U, T)$ from the test chain to the usage chain decreases overall as the number of test cases increases, which proves the fact that the test adequacy enhances gradually with the increase of test cases in software reliability test.

Fig. 3 shows the two matrixes of transition probability of the test chain and the usage chain when the test adequacy is reached. We can obviously discover that the difference between the two matrixes is extremely small, that is to say, the test chain has been very close to the usage chain when the testing process ends.

B. Efficiency Verification of the Test Case Generation Method

In order to test and verify the effectiveness of the test case generation method presented in this paper, an external button is added to our test case generator $MTCG$ to generate test cases in a completely random way. Comparison between the statistical results of test cases generated by Markov usage model and those in a completely random way are made, the results of which indicates that the generation method is efficient.

Table III shows the statistical results of test cases generated in a completely random way for a comparison, including the number of test cases, node coverage, edge coverage and the discriminant value $K(U, T)$. Here $\varepsilon$ is set to 0.00001 and the threshold of $K(U, T)$ is set to 0.001.

<table>
<thead>
<tr>
<th>No.</th>
<th>Number of Cases</th>
<th>Node Coverage</th>
<th>Edge Coverage</th>
<th>$K(U, T)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0.8000</td>
<td>0.5000</td>
<td>0.8055</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1.0000</td>
<td>0.8750</td>
<td>0.6833</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>1.0000</td>
<td>0.8750</td>
<td>0.6258</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>1.0000</td>
<td>0.8750</td>
<td>0.6307</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>1.0000</td>
<td>1.0000</td>
<td>0.5690</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td>1.0000</td>
<td>1.0000</td>
<td>0.3739</td>
</tr>
<tr>
<td>7</td>
<td>20</td>
<td>1.0000</td>
<td>1.0000</td>
<td>0.3048</td>
</tr>
<tr>
<td>8</td>
<td>50</td>
<td>1.0000</td>
<td>1.0000</td>
<td>0.2832</td>
</tr>
<tr>
<td>9</td>
<td>100</td>
<td>1.0000</td>
<td>1.0000</td>
<td>0.0859</td>
</tr>
<tr>
<td>10</td>
<td>200</td>
<td>1.0000</td>
<td>1.0000</td>
<td>0.0968</td>
</tr>
<tr>
<td>11</td>
<td>400</td>
<td>1.0000</td>
<td>1.0000</td>
<td>0.0754</td>
</tr>
<tr>
<td>12</td>
<td>800</td>
<td>1.0000</td>
<td>1.0000</td>
<td>0.0413</td>
</tr>
<tr>
<td>13</td>
<td>1000</td>
<td>1.0000</td>
<td>1.0000</td>
<td>0.0295</td>
</tr>
<tr>
<td>14</td>
<td>2000</td>
<td>1.0000</td>
<td>1.0000</td>
<td>0.0012</td>
</tr>
<tr>
<td>15</td>
<td>2169</td>
<td>1.0000</td>
<td>1.0000</td>
<td>0.0009</td>
</tr>
</tbody>
</table>

Based on the statistical data in Table III, for the same software usage model in Fig. 1, 2169 test cases are generated until the testing process is sufficient in a completely random way. That is to say, in a completely random testing, at least 2169 test cases have to be executed in order to correspond to the actual usage of software system. However, based on the statistical data in Table II, only 391 test cases are generated by Markov usage model when the test adequacy is reached. That is to say, in a Markov usage model-based testing, much fewer test cases (18% in our experiments) need to be executed to correspond to the actual usage of software system. Integrated the comparison and analysis above, we can conclude that the test case generation method based on Markov usage model proposed in this paper is of high efficiency (See Fig. 4).

In addition, we can discover that the testing methods based on Markov usage model can cover all the transitions faster than a completely random testing method if the edge coverage is compared in Table II and Table III. That is to say, software testing based on Markov usage model has stronger ability of path coverage than completely random testing.
which is based on Markov usage model.

Table III. This proves that random testing is much more
According to the third, fourth, ninth and tenth rows in

K(U, T)

C. Characteristic Analysis of Testing Method

Furthermore, we can find that the convergence value K(U, T) from the test chain to the usage chain increases abnormally while the test cases increasing from ten to twenty according to the sixth and seventh rows in Table II. This is a normal phenomenon which is caused by the instability of the test chain. In fact, this instability characteristic is a localized phenomenon, and will not affect the test adequacy determination.

This phenomenon also can be found in Table III. The convergence value K(U, T) increases abnormally twice according to the third, fourth, ninth and tenth rows in Table III. This proves that random testing is much more instable than the testing method proposed in this paper which is based on Markov usage model.

V. CONCLUSIONS

Software reliability test based on Markov usage model is an efficient approach to test cases generation and software reliability evaluation in a quantitative way. In this paper, we propose a modeling method for Markov usage model based on improved state transition matrix (STM), which also contributes to usage model checking. A method for test case generation based on Markov usage model is put forward in detail, and the test adequacy determination method is furtherly presented. We select an improved Kullback discriminant as the convergence judgment criteria from the test chain to the usage chain in order to measure whether the test adequacy has been reached. An experiment with an independently developed test case generation tool MTCG is carried out and the statistical results show that software reliability test based on Markov usage model can generate much fewer test cases (18% in our experiment) than the completely random testing method in the premise of ensuring test adequacy. That is to say, Markov usage model-based software reliability test presented in this paper is approving, high-efficient and promising.

REFERENCES


[29] L. Xu, SMT-based bounded model checking for realtime systems [C], in QSCC, 2008, pp. 120–125.

Kuanjiu Zhou received B.S. degree in 1989 and M.S. degree in 1993 from Harbin Institute of Technology in computer science, Harbin, China, and the Ph.D degree in 1996 from Harbin Institute of Technology in management science and engineering. During his M.S. work, his research interests include computer science, software testing. During his Ph.D work, he specialized in the complexity theory and system engineering. Since 1996, he has been a software engineer, working for Huawei Co. in Shenzhen, and Hisense Group in Qingdao. At Huawei, he leaded a team that developed NSM in an intelligent network. At Hisense, he researched the CDMA (Code Division Multiple Access) mobile. His current research interests include software testing and reliability theory.

Xiaolong Wang received B.S. degree in software engineering in 2011 from Dalian University of Technology, Dalian, China, is a M.S. candidate in computer science at Dalian University of Technology, Dalian, China, since 2011. His current research interests include software engineering, software testing and reliability theory.

Gang Hou received B.S. degree in 2005 from Dalian University of Technology in computer science and engineering, Dalian, China, and M.S. degree in 2008 from Dalian University of Technology in software engineering. His current research interests include software engineering, software testing and reliability theory.

Jie Wang received B.S. degree in 2002 and M.S. degree in 2004 from Harbin Institute of Technology in computer science, Harbin, China, and the Ph.D degree in 2009 from Harbin Institute of Technology in computer architecture. His current research interests include FPGA, software testing and reliability theory.

Shanbin Ai received B.S. degree in software engineering in 2011 from Dalian University of Technology, is a M.S. candidate in computer science at Dalian University of Technology, Dalian, China, since 2011. His current research interests include software engineering, software testing and reliability theory.