ELSEVIER

Information and Software Technology 42 (2000) 889–894

INFORMATION AND SOFTWARE TECHNOLOGY

www.elsevier.nl/locate/infsof

A Markov chain model for predicting the reliability of multi-build software

J.A. Whittaker[a,*], K. Rekab[b], M.G. Thomason[c]

[a]*Software Engineering Program, Florida Tech, Melbourne, FL 32901 USA*
[b]*Department of Mathematical Sciences, Florida Tech, Melbourne, FL 32901 USA*
[c]*Department of Computer Science, University of Tennessee, Knoxville, TN 37996 USA*

Abstract

In previous work we developed a method to model software testing data, including both failure events and correct behavior, as a finite-state, discrete-parameter, recurrent Markov chain. We then showed how direct computation on the Markov chain could yield various reliability related test measures. Use of the Markov chain allows us to avoid common assumptions about failure rate distributions and allows both the operational profile and test coverage of behavior to be explicitly and automatically incorporated into reliability computation.

Current practice in Markov chain based testing and reliability analysis uses only the testing (and failure) activity on the most recent software build to estimate reliability. In this paper we extend the model to allow use of testing data on prior builds to cover the real-world scenario in which the release build is constructed only after a succession of repairs to buggy pre-release builds. Our goal is to enable reliability prediction for future builds using any or all testing data for prior builds.

The technique we present uses multiple linear regression and exponential smoothing to merge multi-build test data (modeled as separate Markov chains) into a single Markov chain which acts as a predictor of the next build of testing activity. At the end of the testing cycle, the predicted Markov chain represents field use. It is from this chain that reliability predictions are made. © 2000 Elsevier Science B.V. All rights reserved.

Keywords: Software failure; Software reliability model; Software testing

1. Introduction

Software reliability estimates are generally made by building probability models of data collected during testing. Testing data is comprised of two different types of events: success events are instances in which the software correctly processes input and *failure events* are situations in which the software exhibits out-of-spec behavior, e.g. premature termination, erroneous computation or mishandling of stored data.

In many cases, software reliability models represent success events only implicitly, usually as they pertain to some definition of *time*. Examples are measuring the number of inputs correctly processed (without tracking exactly which inputs) or keeping track of the so-called wall clock time of failure-free runs. Failure events are treated by assuming that they obey a specific probability law and using that assumption to make estimations and predictions about future failure occurrences.

In some cases, such treatment has delivered valuable and accurate analysis [4]—particularly when the assumed distribution governing failure events has empirical support and substantial testing has covered much of the application's functionality. But without empirical data and the resources to perform substantial testing, other approaches are needed. We proposed in Ref. [14] to model success and failure events as a finite-state, discrete-parameter, recurrent Markov chain. The Markov model has wide applicability as shown in several subsequent case studies [1,3,5,7,9,10]. The model encapsulates success events in advance of testing as states and state transitions in a Markov chain. Thus, one can apply a series of tests and maintain counts of state transitions, providing convenient bookkeeping for test coverage and progress. Obviously, the more complex an application, the larger the state set, requiring a larger test set to properly cover. Missing test inputs and sections of behavior which are poorly covered can be detected by analysis of the model [14].

Failure events are associated with the last known success

* Corresponding author.
E-mail addresses: jw@se.fit.edu (J.A. Whittaker), rekab@cs.fit.edu (K. Rekab), thomason@cs.utk.edu (M.G. Thomason).

0950-5849/00/$ - see front matter © 2000 Elsevier Science B.V. All rights reserved.
PII: S0950-5849(00)00122-1

state and integrated directly into the Markov model as a special *failure state*, say state s_f, such that out-of-spec software behaviors cause transitions to s_f. Each transition to s_f marks a distinct failure event and may differ substantially in its probability of occurrence. This is a desirable characteristic of the model since each failure event is likely to occur at a different rate in the field. Thus, all failures are not treated the same. Their individual probability of occurrence (which reflects their contribution to unreliability) can be computed from the chain. In Ref. [14] we derive equations for reliability, *MTTF*, and a stochastic test stopping criteria.

Thus, the Markov chain may alleviate the need for some assumptions common in reliability models that do not explicitly incorporate software behavior into their formulae. Moreover, recent advances in automating the generation of states and state transitions have made the modeling process practical for increasingly larger systems [2,10]. The contribution of this paper is to combine past and current testing data into a single Markov chain representing predicted testing results for the next build of the software. This will allow us to detect patterns in the way failure events occur and recur and use these patterns to predict future failure events.

Sections 2 and 3 define Markov chains for software testing and discuss their construction. Sections 4 and 5 present a method for statistical analysis of prior testing chains, each of which represents a specific build of the software during testing. Section 6 describes an example use of our new method wherein we predict known Markov chains and compare the accuracy of our predicted chains with the actual chains.

2. A Markov chain model for software testing

A Markov chain model for software testing consists of:

- the finite set S of operational states of a software system;
- the finite set I of externally generated inputs to the software; and
- the probability-weighted transition function δ: $S \times I \times [0, 1] \rightarrow S$.

Operational states describe internal or external objects that influence the behavior of the software under test. In general, we are interested in objects that affect the way the software reacts to external stimuli. We say that software is in state s_j when a collection of objects have certain values and in a different state s_k when at least one of those objects has a different value. The current state also designates which external inputs are allowed and which are prevented (or, at least reacted to in a different manner) by the software. State s_k may have a different set of allowable inputs that mark it as distinct from s_j. Two special states s_{inv} and s_{term} are established to represent invocation and termination of the software and are the initial and final states of the Markov chain, respectively.

The transition mapping δ describes probabilistic changes of state as a function of internal state and application of external input. The model's probability distributions are established through collection of frequency counts during testing. Before testing begins, each transition is set to frequency count 0 to reflect the fact that the software has yet to receive any input. As inputs are applied by testers, the corresponding transitions are identified and their frequency counts incremented to maintain an accurate record of transition coverage during testing. This can be accomplished by instrumenting existing test harnesses[1] or by using the Markov chain itself to generate test cases [5,14].

When failures are encountered, new states are integrated into the chain to model these events. Suppose input j is supposed to move the application from state s_i to state s_k but instead causes a failure in the software to be identified. Create a transition from state s_i to fail state s_f with frequency count 1 (assuming that this is the first failure from state s_i) and an outgoing transition from s_f to state s_k, unless the failure caused the system to halt in which case the outgoing transition is made to the terminate state s_{term}. Additional failures are modeled by establishing new transitions to s_f and failure recurrence is modeled by incrementing existing counts on the appropriate transition.

Whenever estimation using the model is required, the frequency counts can be normalized to probabilities by dividing each frequency by the sum of the outgoing frequencies for each state $s \in S$. When probabilities are assigned to transitions, the model is a finite state, discrete parameter Markov chain and many standard analytical results can be computed without additional assumptions [6]. Moreover, the distribution associated with failure occurrence is embedded in the Markov chain and arises from the statistics associated with real testing data.

3. Modeling software repair

In previous work [14] we modeled software-testing data with a single Markov chain. A problem not addressed, however, was continuity of the model when the underlying software is modified in an effort to repair one or more faults. If the software is changed to eliminate one or more faults, we expect the new build to have fewer embedded defects than previous builds; however, a "repair" that changes the software may in fact result in any of the following:

1. The repair may be successful with no unwanted side-effects. This is the perfect repair situation and allows us to reset to 0 the frequency on the affected transition(s) to the failure state s_f. The new build of the software is an unambiguous improvement over the old.
2. The repair may be locally successful but, as a side-effect, cause a part of the software once deemed fault-free now to be faulty. The implication for the test Markov chain is that, while the transitions to s_f associated with the repair

[1] Web applications can be easily instrumented with page counters to automate this task.

can be reset, new transitions with nonzero frequency count must also be established.
3. The repair may be unsuccessful but without unwanted side-effects. This situation is status quo: there is no improvement overall in the software but no degradation either. The test Markov chain remains unchanged.
4. The repair may be unsuccessful and also introduce unwanted side-effects. In this case the relevant, existing transitions are unchanged and new transitions must be created to represent the side-effects. There is degradation of the software in this unfortunate situation.

There is no way to determine which of these four situations has occurred without further analysis, usually in the form of additional testing. By resetting the frequency counts each time a new build is compiled, our original model took the most conservative approach.

An alternative approach suggested by Walton et al. [12] is to continue incrementing frequency counts across software builds so that one may estimate various aspects of the development process for those builds. However, they too suggest resetting frequency counts for product-oriented estimation such as reliability.

We propose the following method of combining results obtained during additional testing with results from previous testing. The objective is to incorporate the entire testing-and-repair history into a tractable model based on practical, but rigorous, statistical analysis of any or all test data.

Let the initial testing chain T_1 be established and updated with frequency counts and failure state as described above until a repair is attempted. At this time no further change to T_1 is allowed and T_2 is established when testing resumes on the modified software. Consecutive testing chains T_3, T_4,… are similarly constructed.

Thus we establish a single and separate testing chain for each build of the software. The chain T_{k+1} represents testing of the next software build (or usage of the software after release) and has unknown transition probabilities. We are interested in predicting the parameters of this chain using any or all of the chains $T_1,\ldots,T_k$.

Since current state of the practice is to use only the most recent chain T_k to achieve this result [5,12], we believe that improvement in predictive accuracy can result. Since modifications to fix faults have changed T_k, there is every reason to believe (or to hope) that T_{k+1} will differ from T_k in general. Thus, we propose that an analysis of the trends in changes to the transition probabilities over the series $T_1,\ldots,T_k$ will potentially increase the accuracy of our prediction of T_{k+1} rather than using T_k alone.

4. Estimating the parameters of T_{k+1}

Let T^*_{k+1} denote the predictor of T_{k+1}. Obviously, predicting T_1 is not possible since no testing data exists. Likewise, predicting T_2 is problematic since there is only one prior data point and no patterns or trends will have surfaced in the data. However, T^*_3 can be established by analysis of T_1 and T_2. Once this has been accomplished T_4 can be predicted using the real data T_3 combined with its prediction, T^*_3 which contains the history data and trends established from prior testing. Thus, we carry historical data along in the form of our prediction chains throughout the testing process. Note also that since we make predictions for each intermediate build and then later establish actual data, we may discard inaccurate predictions and get feedback on the quality of the predictive process.

Our research suggests that a good prediction of T_{k+1} should involve two prediction techniques: *multiple linear regression* and *exponential smoothing*. Recall that our predictor of T_{k+1} is denoted by T^*_{k+1} and has the form

$$T^*_{k+1} = \alpha^* T_k + (1-\alpha^*)T^*_k$$

where T^*_k is a "good predictor" of T_k based on the transition matrices of the previous chains $T_1, T_2, \ldots, T_{k-1}$.

Specifically, let $\hat{T}_k$ be a predictor of T_k determined via multiple linear regression and let $\tilde{T}_k$ be a predictor of T_k determined through the application of exponential smoothing. $\tilde{T}_k$ will depend on α^*, a suitable constant satisfying $0 < \alpha^* < 1$ and

$$\sum_{i=3}^{k} d(\tilde{T}_i(\alpha^*), T_i) = \min_{0<\alpha<1} \sum_{i=3}^{k} d(\tilde{T}_i(\alpha), T_i),$$

(see Section 6.1 for more details).

T^*_k is chosen so that

$$d(T^*_k, T_k) = \min\{d(\hat{T}_k, T_k), d(\tilde{T}_k, T_k)\}$$

where $d(T^*_k, T_k)$ denotes a distance from transition matrix T^*_k to matrix T_k. We will use a distance measure of the form

$$d(T^*_k, T_k) = \sum_m \sum_n \frac{|(T^*_k(m,n) - T_k(m,n))|}{T_k(m,n) + 1}$$

where m is the matrix-row index and n is the matrix-column index. This distance allows for a relative difference instead of an absolute difference. See Ref. [11] for more details.

Thus, we determine the parameters of T^*_{k+1} using the previous T_k established with real testing data and T^*_k which is a predictor of T_k encapsulating the history of chains $T_1,\ldots,T_{k-1}$. As we continue testing, we will eventually get the actual T_{k+1} and can compare it to our estimate T^*_{k+1} using the distance measure above. In this manner we can track the accuracy of our predictions through each successive build of the software. In fact, the distance measure may indicate that certain builds of testing chains may be atypical for one reason or another, allowing us to drop those data points from our analysis.

The ultimate goal is to determine T^*_{k+1} for which no T_{k+1} will be known, i.e. where T_k represents the final build before release. Thus, T^*_{k+1} becomes our prediction of field usage and estimates computed from it represent post-release reliability.

Table 1
Selection of α for exponential smoothing

α	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
$D(\alpha)$	270	228	201	175	174	167	163	152	161

5. Predictors of the transition matrix T^*_{k+1}

We use one of two methods to predict T^*_{k+1} based on the transition matrices $T_1, T_2, \ldots, T_{k-1}$. $\tilde{T}_k$ is derived using *exponential smoothing* and $\hat{T}_k$ is derived using *multiple linear regression*. The one, which provides the smaller distance measure, as defined above gets selected.

5.1. Derivation of $\tilde{T}_k$ (exponential smoothing)

$\tilde{T}_k$ is determined through the recursion

$$\tilde{T}_k = \alpha T_{k-1} + (1-\alpha)\tilde{T}_{k-1}$$

$$\tilde{T}_{k-1} = \alpha T_{k-2} + (1-\alpha)\tilde{T}_{k-2}$$

$$\vdots$$

$$\tilde{T}_3 = \alpha T_2 + (1-\alpha)\hat{T}_2$$

where $\hat{T}_2$ is an estimate of T_2 determined by linear regression.

5.2. Derivation of $\hat{T}_k$ (multiple linear regression)

The general form of $\hat{T}_k$ is

$$\hat{T}_k = a_0 + \sum_{i=1}^{k-1} a_i T_i + \sum_{i<j} \sum_{j=2}^{k} a_{ij} T_i T_j$$

where a_i for $i = 1, \ldots, k-1$ and a_{ij} for $i < j, j = 1, \ldots, k$ are determined through the least square method. In certain cases, the value of the distance is not increased significantly by including the interaction terms $T_i T_j$. In these cases, a simple multiple-linear regression model of the form

$$\hat{T}_k = a_0 + \sum_{i=1}^{k-1} a_i T_i$$

may be used.

Table 2
Regression model relating T_9 with $T_1, \ldots, T_8$

	Unstandardized coefficients	
	B	S.E.
(Constant)	-0.1×10^{-3}	0.141
T_1	−1.150	0.439
T_2	0.534	0.284
T_3	2.259	0.282
T_4	−1.308	0.287
T_5	0.378	0.074
T_6	−1.453	0.221
T_7	0.865	0.110
T_8	1.200	0.180

6. Example

We have data from a small development effort conducted under industrial contract. During this development, a 12-state Markov chain was constructed to generate test cases and ten iterated builds of the software occurred before its release. A single failure state s_f was created in the chains as a thirteenth state for the transitions due to software faults. We are interested in using the first nine chains to predict the tenth. Since the tenth chain is known, we can compare the accuracy of our predictions. Given $T_1, T_2, \ldots, T_9$ as the nine chains for the respective pre-release builds, we seek to determine T^*_{10} as a suitable predictor of T_{10}.

6.1. Exponential smoothing

As described in Section 4, the predictor of chain T_9 has the form

$$\tilde{T}_9(\alpha) = \alpha T_8 + (1-\alpha)\tilde{T}_8$$

$$\vdots$$

$$\tilde{T}_3(\alpha) = \alpha T_2 + (1-\alpha)\hat{T}_2$$

$$\hat{T}_2 = 0.129 + 1.561 T_1$$

where $\hat{T}_2$ is a predictor of T_2, given T_1, determined through simple linear regression.

$\tilde{T}_9(\alpha^*)$ is chosen so that

$$\sum_{i=3}^{9} d(\tilde{T}_i(\alpha^*), T_i) = \min_{0<\alpha<1} \sum_{i=3}^{9} d(\tilde{T}_i(\alpha), T_i)$$

where the distance d is defined as in Section 4. Table 1 represents the values of $D(\alpha) = \sum_{i=3}^{9} d(\tilde{T}_i(\alpha), T_i)$. The minimum occurs for $\alpha^* = 0.8$ giving

$$\tilde{T}_9 = 0.8 T_8 + 0.2 \tilde{T}_8.$$

Also, note that

$$d(\tilde{T}_9, T_9) = \sum_{m=1}^{13} \sum_{n=1}^{13} \frac{|(\tilde{T}_9(m,n) - T_9(m,n))|}{T_9(m,n) + 1} = 13.$$

6.2. Multiple linear regression

Multiple linear regression relates transition matrix T_9 to $T_1, \ldots, T_8$ in one linear functional form. Results in least squares coefficients are presented in Table 2.

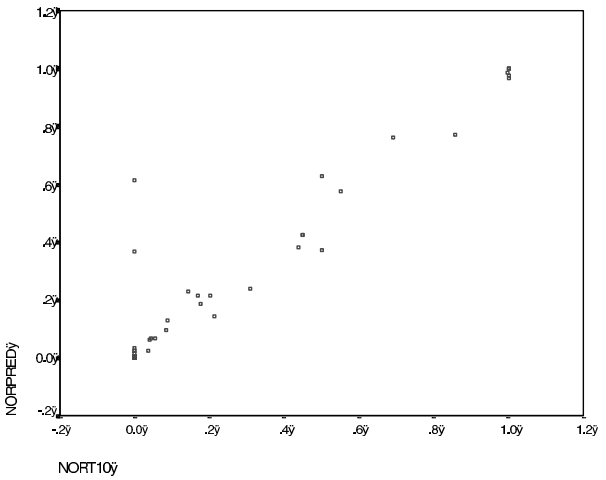

Fig. 1. Scatter plot: normalized predicted vs. normalized observed for transition matrix T_{10}.

The prediction equation is

$$\hat{T}_9 = -0.003 - 1.150T_1 + 0.534T_2 + 2.259T_3 + 1.308T_4 + 0.378T_5 - 1.453T_6 + 0.865T_7 + 1.200T_8,$$

and

$$d(\hat{T}_9, T_9) = \sum_{m=1}^{13} \sum_{n=1}^{13} \frac{|(\hat{T}_9(m,n) - T_9(m,n))|}{T_9(m,n) + 1} = 16.32$$

which is larger than $d(\tilde{T}_9, T_9)$. The better predictor, therefore, of T_9 is $\tilde{T}_9$. Consequently, the predictor of T_{10} is $T^*_{10} = 0.8T_9 + 0.2\tilde{T}_9$.

6.3. Adequacy of the predicted transition matrix

In our data, T_{10} contained no failures. This is common in some projects where release is forbidden unless a build tests clean. However, predicting no field failures has little value since it is unlikely that testing covered every possible input scenario combination.

This is a case in which our new method can predict failure likelihood in the absence of failures in the current build. Since it uses both current build data and prior build data, we have information about past failures and the failure occurrence trend from prior builds.

In order to compare our prediction T^*_{10} to the observed results T_{10}. We normalized each row of both transition matrices and performed exponential smoothing and multiple linear regression, accepting the best predictor for each successive build. Fig. 1 is a scatter plot for the normalized predicted matrix T^*_{10} (shown as the y-axis) vs. the normalized observed matrix T_{10} (shown as the x-axis). Most cases in the plot show that the predicted transition matrix is very close to the observed transition matrix (indicated visually by clustering along the diagonal). The two points that stray furthest from the diagonal correspond to failure states in chain T^*_{10}. Since T_{10} contained no failures, the failure values in T^*_{10} were compared to zero. Obviously, more testing would bring these two values closer to the diagonal.

7. Conclusion

We presented a technique to combine multi-build testing results each of which are modeled as separate Markov chains to predict testing results for the next software build. When such predictions are made using the last build before release, the predictions represent expected field use. Reliability analysis (presented in another paper [14]) on the predicted chains can be used to quantify software behavior in the field.

Currently, we are performing more substantial field trials under a contract from IBM Storage Systems Division in Tucson, Arizona. We hope to present this data in future papers.

Acknowledgements

The authors have had many helpful discussions with practitioners, most especially Mike Houghtaling of IBM and Kaushal Agrawal of Cisco. We would also like to thank the anonymous referees for many helpful suggestions.

References

[1] K. Agrawal, J.A. Whittaker, Experiences in applying statistical testing to a real-time embedded software system, Proc. Pac. Northwest Soft. Quality Conf., Portland OR, 1993, pp. 154–170.
[2] I. El-Far, Automated Construction of Markov Chains for Software Testing, MS Thesis, Comp. Sci. Program, Florida Inst. of Tech, Melbourne, FL 1999.
[3] O. Dahle, Statistical Usage Testing Applied to Mobile Telecommunications Systems, MS Thesis, Dept of Comp. Sci., Univ. of Trondheim, Norway, 1995.
[4] W. Farr, Software reliability modeling survey, in: M.R. Lyu (Ed.), Handbook of Software Reliability Engineering, IEEE Computer Society Press, New Jersey, 1996, pp. 71–115.
[5] M. Houghtaling, Automation frameworks for Markov chain statistical testing, Proc. Automated Software Test Evaluation Conf., EFPDMA Press, Washington DC, March 1996.
[6] J.G. Kemeny, L.J. Snell, Finite Markov Chains, Springer, New York, 1976.
[7] R. Oshana, Tailoring cleanroom for industrial use, IEEE Software 15 (6) (1998) 46–55.
[9] M. Rautakorpi, Application of Markov Chain Techniques in Certification of Software, MS Thesis, Dept of Mathematics and Systems Analysis, Helsinki Inst. Tech., Helsinki, Finland, 1995.

[10] H. Robinson, Finite model-based testing on a shoestring budget, Proc. Software Testing Analysis Review Conf., San Jose, CA, Nov. 1999.
[11] V.K. Rohatgi, An Introduction to Probability Theory and Mathematical Statistics, Wiley, New York, 1976.
[12] G.H. Walton, J.H. Poore, C.J. Trammell, Statistical testing of software based on a usage model, Software Practice and Experience 25 (1) (1993) 98–107.
[14] J.A. Whittaker, M.G. Thomason, markov chain model for statistical software testing, IEEE Transactions on Software Engineering 20 (10) (1994) 812–824.