# Load Testing Software using Deterministic State Testing

Alberto Avritzer
AT&T Bell Laboratories
Red Hill, NJ 07748-3052 *

Brian Larson
AT&T Bell Laboratories
West Long Branch, NJ 07764-1394 †

**Abstract**

In this paper we introduce a new load testing technique called Deterministic Markov State Testing and report on its application. Our approach is called 'deterministic' because the sequence of test case execution is set at planning time, and 'state testing' because each test case certifies a unique software state. There are four main advantages of Deterministic Markov State Testing for system testers: provision of precise software state information for root cause analysis in load test, accommodation for limitations of the system test lab configuration, higher acceleration ratios in system test, and simple management of distributed execution of test cases. System testers using the proposed method have great flexibility in dealing with common system test problems: limited access to the system test environment, unstable software, or changing operational conditions. Because each test case verifies correct execution on a path from the idle state to the software state under test, our method does not require the continuous execution of all test cases. Deterministic Markov State Testing is operational-profile-based, and allows for measurement of software reliability robustness when the operational profile changes.

## 1 Introduction

Deterministic Markov State Testing is a test-case-generation and execution approach for testing telecommunication software according to the operational profile. However, our approach does not sample the test cases randomly from the operational distribution. Instead, the operational profile is used to build a Markov chain [1] that represents the software's behavior.

In our telecommunication system, the Markov chain state at time $t$ consists of the number of calls of each type being processed at time $t$. Each Markov chain state is a possible test case. In load testing our system, it is unfeasible to test all Markov states. Each test case requires the establishment of a telephone call and the holding of a channel for the whole duration of the telephone conversation: we assume that an average phone call takes three minutes. For a telecommunication system with 24 channels and five types of calls the total number of Markov states is 118,755, the number of ways to distribute five balls into 24 urns [7]. To test all states would require 1484 four-hour testing sessions.

Our test-case-generation approach is based on the operational profile because only the most likely tests, as computed from the Markov chain solution, are generated. To test the most likely software states[2], 40 four-hour sessions were required, thus providing a 37-fold reduction in the length of the load test phase.

Our execution approach is deterministic in the sense that a sequence of test cases is generated at planning time; i.e. before the start of system test. The list of test cases is built from the most probable Markov chain states. Each test case certifies a unique software state[3].

The purpose of system testing is to detect system-wide failures that are the result of interactions among software components. System testing is concerned with performance, security, reliability, configuration sensitivity, start-up, recovery, and accountability[1]. Load testing is a sub-phase of the system test phase composed of the two activities related to the expected workload/operational profile in the field: performance testing and reliability testing.

The deterministic execution of test cases in load test provides the following advantages to system testers:

System testers can account for limitations of the system test hardware configuration by preprocessing the list of test cases and removing those that are not executable in the test configuration. The impact of the limited hardware configuration on the quality of testing is a function of how often the excluded tests occur in field operation.

Because each software state is validated independently, when a failure is detected, system testers can precisely tell developers which software state was last executed and what sequence of software states led to the failure.

In random load testing, arrival and service times of requests are usually drawn from a probabilistic distribution. The variability of the arrival and departure processes is used to build the system load. In contrast, we use deterministic

*Author's address Room 2E-030, 480 Red Hill Road, Middletown, NJ 07748-3052, beto@eagle hr.att com

†Author's address.Room 3E-125, , West Long Branch, NJ 07764-1394, larson@homxb att com

[1] A Markov chain is a method for computing test case occurrence probabilities

[2] In this work each software state corresponds to a Markov chain state.

[3] In this analysis we are concerned with the macro resources consumed by call processing in telecommunication systems daemons, buffers, database locks, message queues, etc



**ACM-ISSTA'93-6/93/Cambridge, MA, USA**

arrival and departure processes. For each software state, we build the load from the idle state by initiating the precise number and type of requests represented in the software state under test. Because we wait for the previous test case to finish before starting a new one, we can run the load test machine at near 100% CPU utilization without causing resource queues to grow without bound. The practical test acceleration factors in random load test are limited, because most software is designed to be operated at medium to low utilizations. However, load test productivity requires the software to be run at medium to high utilizations.

If the test environment is a distributed system, i.e. multiple machines connected by a local area network, the list of test cases can be divided among the system test machines.

Deterministic Markov State testing uses the operational profile and a Markovian state definition to generate a continuous-time Markov chain[4], whose steady-state probability solution is used to guide the generation of test cases. Any unit testing approach[1, 4] can be used to generate the proper input variable value for each test case. The test cases have a weight given by the steady-state probability solution and therefore reflect the statistical use of the software. Moreover, to make the best possible use of the testing resources, test cases are executed deterministically, one after the other. If the operational profile changes, a new Markov chain is generated and solved. However, only the difference between the new test case list and the original test case list needs to be run.

We use established models[6] to estimate the software reliability, as failures are detected. Software failures are associated with software states. To compute the calendar time needed in the field to execute a corresponding number of calls, we perform an execution to calendar time transformation.

The remainder of this paper is as follows. In Section 2 we describe our system. In Section 3 we derive the system operational profile. In Section 4 we define the Markov chain and describe its solution. In Section 5 we describe the test-case-generation algorithm. In Section 6 we describe the test case execution approach. In Section 7 we describe the execution to calendar time transformation. Section 8 contains our conclusions.

## 2 The System

Deterministic state testing was applied to a future telecommunication service offering. Some details have been modified to protect proprietary information.

The system will enable a caller to reach a subscriber to the service by dialing a unique Personal Telephone Number (PTN). Subscribers will be able to specify and easily change a list of call-forwarding telephone numbers. The call-forwarding profile is consulted whenever a caller tries to reach a subscriber.

## 3 Operational Profile

The operational profile for the system was derived for its Software Reliability Engineering. We identified five types of calls:

1. Client call completes to subscriber,
2. Client call cannot reach subscriber, and therefore goes to voice mail,
3. Subscriber call to update subscriber profile,
4. Subscriber call to check status,
5. Subscriber call to retrieve voice mail messages.

The project estimated that client calls are 0.737 of the total and subscriber calls are 0.263; 0.80 of the client calls complete to subscriber and 0.20 cannot reach the subscriber, and therefore go to voice mail; 0.40 of the subscriber calls were estimated to be updates, 0.40 voice mail access, and 0.20 status checking. The following is the operational profile for our system:

1. Client call completes to subscriber = 0.737*0.8 = 0.589,
2. Client call cannot reach subscriber, and therefore goes to voice mail = 0.737*0.20 = 0.1474,
3. Subscriber call to update subscriber profile = 0.263*0.40 = 0.1052,
4. Subscriber call to check status = 0.263*0.20 = 0.0526,
5. Subscriber call to access voice mail = 0.263*0.40 = 0.1052.

## 4 Markov Chain

### 4.1 Description and Assumptions

Table 1 contains a summary of the assumptions used to define the Markov chain. Our method builds a Markov chain [5] for the system from the operational profile's average call arrival rate and holding time[5] for each call type. In this work we consider a universe of 500 subscriber accounts. Each handles on average 16 calls in a 24-hour period. The specification for the average arrival rates in calls-per-minute per call type is presented in Table 1. We assume that each call type forms an independent Poisson process with the specified arrival rate. For Markov processes the times between state transitions are exponentially distributed [5]. Therefore, to map our state definition into a Markov process we need the additional assumption that each call type has a holding time that is distributed exponentially. The estimated average call holding times are also presented in Table 1.

The system will operate at low CPU utilizations because the system requirements specify that the maximum probability of call blocking is 0.001. Therefore, we assume that queueing for the CPU and the database will be negligible, with calls dropped (caller gets busy tone) when all channels are busy. As a consequence of this assumption, we get the following approximation: the average call service time depends only on the call type and not on the number of calls in process.

[4] In continuous-time Markov chains transitions occur at any time In contrast, in discrete-time Markov chains, transitions occur only at discrete times.

[5] Holding time for a call is defined as the time during which the call is active: a call is being established or has a channel allocated to it. It therefore includes all the time the client or subscriber is using the system.

| Call Type | Average Arrival Rate (calls/min.) | Average Holding Time (min.) | Average # of calls in process of each type | Arrival process | Holding time distribution |
|---|---|---|---|---|---|
| Client call completes to subscriber | 3.3 | 3.0 | 9.9 | Poisson | Exponential |
| Client call cannot reach subscriber | 0.823 | 3.0 | 2.46 | Poisson | Exponential |
| Subscriber call to update subscriber profile | 0.588 | 2.0 | 1.17 | Poisson | Exponential |
| Subscriber call to check status | 0.297 | 1.0 | 0.297 | Poisson | Exponential |
| Subscriber call to access voice mail | 0.588 | 5.0 | 2.94 | Poisson | Exponential |

Table 1: Call Traffic Assumptions

### 4.2 State Definition

To describe the Markov chain generated for our system, we now introduce the Markovian state concept. Let $S(t)$ be the Markovian state $S$ at time $t$. $S(t)$ is defined to be Markovian if it summarizes the whole system's past history. Therefore, any state $S(r)$ that can be reached from state $S(t)$ through a one-step transition depends on the past only through state $S(t)$. This is called the memoryless property. The only continuous-time distribution that exhibits the memoryless property is the exponential[5]. Therefore, a state will be Markovian, with respect to the time of state transitions, if the average time to a transition out of the state is exponentially distributed.

In this study, we consider two types of transitions for our system: call arrivals and call completions. We have assumed that each call type forms a Poisson arrival process with interarrival times that are exponentially distributed. Also, call holding times were assumed to be exponentially distributed.

Let the number of calls of each type being processed by the system at time $t$ be:

$S(t)$ = (number of calls of type 1 being processed at time t, number of calls of type 2 being processed at time t, number of calls of type 3 being processed at time t, number of calls of type 4 being processed at time t, number of calls of type 5 being processed at time t).

$S(t)$ defines a Markovian state, because calls will arrive and depart from the system at times that are sampled from the exponential distribution.

### 4.3 State Transitions

We start by describing transitions out of and back to the idle state. When the system is idle, it is in the state $S(t)$ = (0,0,0,0,0). Figure 1 illustrates the relationship between call rates and state transitions. The rates shown are in calls/minute. The transition rate from a Markov chain state back to the idle state is the inverse of the average holding time of the call type shown in Table 1[6]. Because the arrival process is independent of the state that the system is in at time t, calls of any type can arrive at any state. However, since our test machine can process a maximum of 24 calls in parallel, and has no buffer space for waiting calls, arrivals at states with 24 outstanding calls will be dropped; i.e., calls that arrive to find all channels busy will receive a busy tone. Therefore, there will be no transitions out of any state where the sum of outstanding calls is equal to 24. Hence, we have a total of 118,755 states. Each state in the Markov chain represents a possible test case. However, it is too expensive to test software states with a very low field occurrence probability. The steady-state probability solution for the system's Markov chain provides the mathematical basis for selecting test cases.

### 4.4 Steady State Solution

Some Markov chains that represent birth-death queueing systems have their steady-state probability already tabulated [5]. Our approach will be to decompose the Markov chain steady-state probability into factors that can be approximated by the solution of a system with Poisson arrivals, exponentially distributed departures, 24 simultaneous calls processed in parallel, and queuing space for 24 calls ($M/M/24/24$) [5]:

$$P_{N_x} = \begin{cases} P_0 \left(\frac{\lambda_x}{\mu_x}\right)^{N_x} \frac{1}{N_x!} & N_x \leq 24 \\ 0 & N_x > 24 \end{cases} \quad (1)$$

In equations (1) and (2), $P_{N_x}$ is the steady-state probability of having $N_x$ calls of type $x$ , $\lambda_x$ is the arrival rate of the call type, $\mu_x$ is the completion rate of the call type, and $P_0$ is the probability of an idle system (no calls of the given type are in process).

We now introduce an independence assumption[7] between call types that yields an approximation for the aggregate solution of the Markov chain. We assume that our Markov chain is nearly completely decomposable [3]; that is, for each call type we have an M/M/24/24 loss-queueing-system solution. The joint probability distribution is the product of the individual distributions: we assume a product-form solution. To fit the solution to the Markov chain, we discard any state with more than 24 outstanding calls.

[6] In states with multiple calls in process, each call type will have a completion rate that is the product of the number of calls of the type, in the state, times the completion rate of the call type

[7] An independence assumption to yield a product-form solution for models representing packet switching networks was introduced by Kleinrock, and is known as the Kleinrock independence approximation [2].

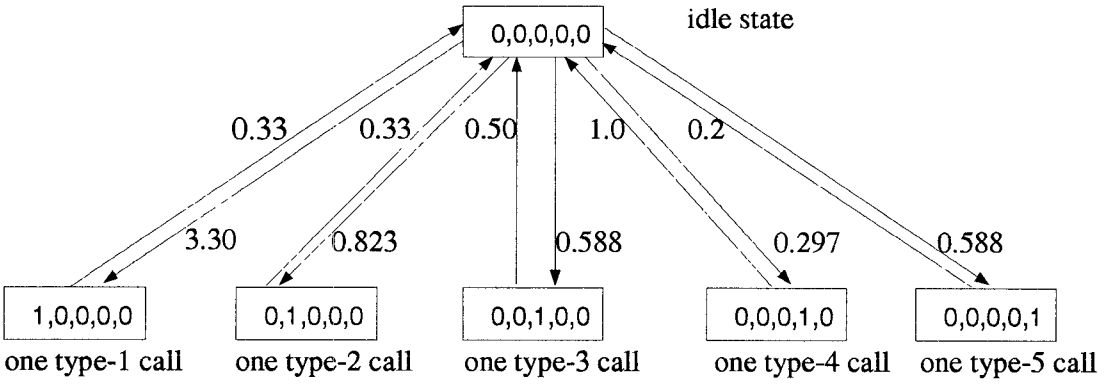


Figure 1: Transitions out of and back to the idle state in calls per minute

Our approximation yields the following steady-state probability for state $S = (N_1, N_2, N_3, N_4, N_5)$,

$$P(N_1, N_2, N_3, N_4, N_5) = P_0 \prod_{x=1}^{5} \left(\frac{\lambda_x}{\mu_x}\right)^{N_x} \frac{1}{N_x!} \qquad (2)$$

In equation (2), $N_x$ is the number of outstanding calls in the state for call type $x$. $P_0$, which is the normalization constant for the steady state distribution, can be computed from the requirement that all state probabilities must sum to one.

The justification for our independence assumption comes from the following system requirements:

1. The system will operate at low CPU utilizations (probability of blocking in the requirements is 0.001).

2. Queueing for the CPU and the database will be negligible, with calls dropped (caller gets busy tone) when all channels are busy. Therefore, the average call service time depends only on the call type and not on the number of calls in process, since each call has its own channel, and most of the call-holding time is allocated to client/subscriber communication. The interaction in processing occurs during call set-ups and profile checks/updates, since all calls share the system's computer and database. However, most calls are client/subscriber communication, and set-up time for this type is short. Also, because we operate with a very small probability of call blocking, the call acceptance rate is approximately equal to the call demand rate.

## 5 Test Case Generation

The software steady-state probability distribution is a powerful tool to guide test-case-generation, because it tells us, for a given system's operating time, which fraction of that time the software will expend in each state. Therefore, the most probable states in the Markov chain solution represent software states that are the most likely to be reached in the field.

The failure intensity objective for the system was set as 0.01 failures/CPU hour of execution. To predict the initial failure intensity at the start of system test, we used the parameter-determination technique for the exponential software reliability model as described in [6]. Statistics from similar projects were used to predict the number of faults present in the code at the start of system test and the rate at which those faults were exposed. The initial failure intensity at the start of system test was predicted as 24.2 failures/CPU hour. System test execution time duration was planned as 31.4 CPU hours, using the exponential model. The average call holding time is three minutes, which we assumed to account for about 0.03 minutes of execution time. Therefore, the number of required calls, $M$, is $62,800 = \frac{31.4 \times 60}{0.03}$.

The proposed approach is to generate only the most probable software states, $N$, which have probabilities greater than a heuristically computed $\epsilon$. $\epsilon$ and $N$ are related by the steady-state probability distribution, and by the total number of calls, $M$, generated by the $N$ test cases. The test-case-generation algorithm should start with an $\epsilon$ equal to a small number and tune it until the number of calls generated by the test cases is approximately $M$. Tuning $\epsilon$ can be done using binary search.

We now turn to the algorithm that generates the most probable test cases, $N$. We start by motivating the approach.

Given $p(S)$, the steady-state probability for state $S$, all the states derived from it by the addition of one more call of a generic type $x$ have a probability equal to:

$$p(S + x) = p(S)\frac{\lambda_x}{\mu_x N_x} \qquad (3)$$

According to our assumptions, the system operates at low CPU utilization (below 50%) and the probability of

blocking is less than 0.001. Therefore, for every call type $x$ there are a number of outstanding calls $N_x$ for which $\frac{\lambda_x}{\mu_x N_x} < 1$.

The above equation, when $N_x = 24$, represents the expected number of outstanding calls for the generic call type $x$. To satisfy the blocking probability requirement of 0.001, the number of outstanding calls, $N_x$, that satisfies the above equation must be smaller than 24. Table 1 contains the average number of calls in process for each call type. Therefore, the steady-state probability for states with many outstanding operations is very small. Furthermore, given that we have reached a state that has a steady-state probability distribution less than $\epsilon$, all states generated from it by the addition of operations of type $x$, when $\frac{\lambda_x}{\mu_x N_x} < 1$, will have a lower steady-state probability, because we are multiplying the discarded state probability by a number that is less than one. When we find such a state, we don't need to generate any test cases by adding calls of type $x$.

The structure of the problem leads to a nice recursion: generate the current state and all states derived from it by adding one more outstanding call of type $x$, until the test case probability is less than $\epsilon$. On return, increment to the next type of call, and recursively repeat the procedure.

To simplify the description of the algorithm, we assume that call types are sorted by $\frac{\lambda_x}{\mu_x}$. That is, the call type with the highest expected number of active calls will be call type 1, the second highest will be type 2, and so on, with the lowest expected being type 5.

The algorithm to generate a list of the test cases, starting from an arbitrary state S, is as follows:

**Algorithm T(S): Generate a list of test cases starting from the software state S:**

1. Set the index to call types $x$ to 1.
2. Set $N$ to 1 plus the number of outstanding calls of type $x$ in state S.
3. If by adding one more call of type $x$ we reach a new state, and the steady state probability of the state generated by adding one more call of type $x$ is greater than $\epsilon$; OR $\frac{\lambda_x}{N\mu_x} \geq 1$ then:
   - Generate a test case for the software state reached from S by adding one more outstanding call of type $x$. Call it state $S + x$;
   - Generate a list of test cases by recursively executing T($S + x$);
   - If $x$ is greater than the number of call types (in the example, 5) return;
   - Otherwise set the index of call types $x$, to $x + 1$, and go to step 2.

The algorithm is initialized by setting $S = (0, 0, 0, 0, 0)$, and executing $T(S)$.

The implementation of the test-case-generation algorithm produced a list of vectors with seven elements: the first is the test case number, the next five denote the number of call operations of each type to be performed in this test case, and the last is the computed steady-state probability of this system state.

We reduced the set by eliminating states with probabilities below $\epsilon$, resulting in 3172 test cases, which included 70,018 individual calls.

## 6 Test Case Execution

In this section we describe the approach for test case execution [8]. The software state certification is a two-step procedure:

1. Use the software state value as defined in the test case,
2. Select the input variables values for the software state selected in step 1. For example, if we have a test case with 12 calls from client to subscriber in step 1, in step 2 we have to select 12 subscriber phone numbers. We assume that a method for selecting input variable values has been derived in unit testing.

Our approach assumes that the order in which test cases are executed is not relevant, because in load test we are concerned with the system-wide software resources (e.g., daemons, buffers, message queues) consumed by call processing. Therefore, if we have multiple system-test machines, we can speed up system-test by dividing the test cases among the test machines.

### 6.1 Test Environment

In Figure 2 we show the test environment components:

1. **The Test Engine** - places calls through a local PBX to the system's voice response unit;
2. **The Local PBX Switch** - forwards calls to the voice response unit;
3. **The Voice Response Unit** - queries and updates the subscriber database server, and in the case of client calls, places outbound calls to the subscriber destination number, back through the PBX;
4. **The Subscriber Database Server** - stores account information and billing records, and manages system operations.

To run test cases, we had to develop test script scenarios to initiate calls, to answer calls, and to collect test/fail results for each test case.

### 6.2 Test Script Scenarios

We ran a battery of automated scripts on the test engine to simulate the call types as described in the operational profile, and to accept outbound calls placed by the system under test. The test script scenarios were as follows:

1. Client reaches first destination number - This script is invoked on an outbound voice channel, calling a personal telephone number (PTN). Once connected to the answering number, the call script will hold approximately three minutes, then will terminate by hanging up the call.
2. Client reaches voice mail - This script calls in exactly like the first script, but connects to a voice mail service.

[8] To simplify our algorithms we assume that we can certify test cases to be error free, and therefore we don't need to execute duplicated test cases. In system test we are concerned mostly with resource allocation, which is independent of the path taken to reach a software state. A simple algorithm modification is required to generate duplicated test cases

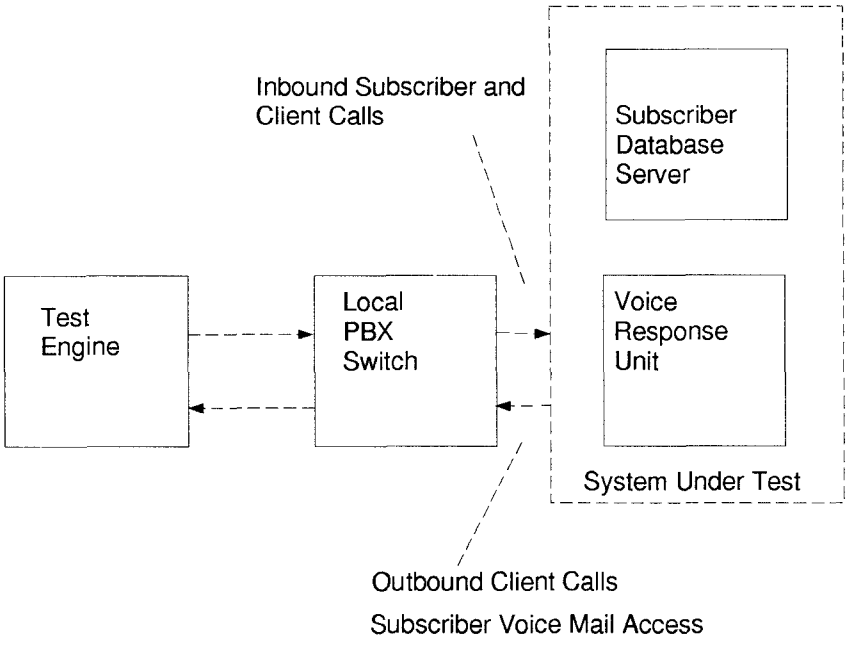


Figure 2: Test Environment

3. Subscriber profile update - All subscribers go to the same 800 number. However, each subscriber has unique account and personal identification numbers. After connecting, the script listens to the system 'welcome message,' selects profile administration, changes system settings, and terminates by hanging up the call after approximately two minutes of elapsed time.

4. Subscriber voice mail access - This script operates like the third script, but after listening to the system 'welcome message,' it selects voice mail access, holds for five minutes, exits voice mail, and terminates by hanging up the call.

5. Subscriber check status - This script also operates like the third script, but after listening to the system 'welcome message,' it listens to a readout of the subscriber's call-forwarding schedule, and terminates by hanging up the call after approximately one minute.

6. Call answering script - This script answers the incoming call from the voice response unit when it is trying to reach a subscriber destination. It will hold the line until the caller script hangs up.

### 6.3 Results

Data from 11 test sessions are shown in Table 2. We show the total number of client and subscriber calls made during the run, and the completion rate achieved. The system under test was rebooted after sessions three and nine. The system test group was able to make recommendations for software quality improvement after the load test phase.

## 7 Execution to Calendar Time Transformation

In our testing approach, we can perform a simple mapping between the number of test cases executed in system test and the expected calendar time required in the field to execute the same test cases. We made the assumption in Section 4 that calls arrive according to independent Poisson processes. Their sum is a Poisson process with a rate equal to the sum of the rates of the component processes[5]. The average number of calls that arrive in time $t$ from a Poisson process of rate $\lambda$ is equal to the product $\lambda t$. Therefore, to compute t, the calendar time needed to receive M calls from the Poisson process, we simply compute $\frac{M}{\lambda}$.

For our system we computed the total number of calls, $M = 62,800$, the total call arrival rate, $\lambda = 5.6/minute$, and the required execution time in system test as 31.4 hours. The calendar time equivalent to the test of 62,800 calls is 11,214 minutes, or 186.9 hours.

## 8 Conclusions

In this work we have introduced a new load testing technique called Deterministic Markov State Testing, and applied it to the system test phase of a future telecommunication service offering. Data from 11 test sessions including over 31,000 calls was collected. We covered 77% of the steady-state probability mass produced by the test-case-generation algorithm. The system test group was able to make recommendations for software quality improvement after the load test phase.

In Deterministic Markov State Testing a continuous-time Markov chain is used to compute which software states are

| Session # | *Client Calls* | | *Subscriber Calls* | |
|---|---|---|---|---|
| | Calls Initiated | Calls Completion Rate (%) | Calls Initiated | Calls Completion Rate (%) |
| 1 | 1516 | 97 | 77 | 99 |
| 2 | 1679 | 48 | 179 | 47 |
| 3 | 1578 | 23 | 261 | 24 |
| 4 | 1547 | 100 | 279 | 96 |
| 5 | 1531 | 99 | 358 | 92 |
| 6 | 1360 | 98 | 402 | 90 |
| 7 | 5869 | 40 | 1328 | 33 |
| 8 | 1280 | 20 | 478 | 26 |
| 9 | 4715 | 22 | 752 | 26 |
| 10 | 2668 | 100 | 511 | 100 |
| 11 | 2775 | 100 | 566 | 98 |

Table 2: Test Results

good test cases. Test cases are generated for the most probable software states. The deterministic execution of test cases in load test provides multiple advantages to system testers.

Because each software state is validated independently, when a failure is detected, system testers can provide precise information to developers on which software state was last executed, and on the sequence of software states that led to the failure.

System testers can account for limitations of the system test hardware configuration by preprocessing the list of test cases and remove those that are not executable in the test configuration. The impact of the limited hardware configuration on the quality of the testing is a function of how often the excluded tests occur in field operation.

System testers can run the load test machine at a much higher CPU utilization than with random execution of test cases. The practical test acceleration factors in random load test are limited by the maximum CPU utilization the system is designed to sustain.

If the test environment is a distributed system, i.e. multiple system test machines connected by a local area network, the list of test cases can be divided among the system test machines.

## References


[1] B. Beizer. *Software Testing Techniques.* Van Nostrand Reinhold, 1990.

[2] D. Bertsekas and R. Gallager. *Data Networks.* Prentice-Hall, 1987.

[3] P. J. Courtois. *Decomposability, Queueing and Computer System Applications.* ACM Monograph Series. Academic Press, 1977.

[4] DeMillo, McCracken, Martin, and Passafiume. *Software Testing and Evaluation.* Benjamin/Cummings Pub., 1987.

[5] L. Kleinrock. *Queueing Systems, Volume I: Theory.* Wiley-Interscience, New York, 1975.

[6] J. D. Musa, A. Iannino, and K. Okumoto. *Software Reliability Measurement, Prediction, Application.* McGraw-Hill, 1987.

[7] K. S. Trivedi. *Probability and Statistics with Reliability, Queuing, and Computer Science Applications.* Prentice-Hall, 1982.


## 9 Acknowledgments


We would like to thank Elaine Weyuker, Kay Smith, Willa Ehrlich, John Musa, Bill Everett, Larry Votta, and the anonymous referees for their suggestions for improvement of this work.